Univerzita Karlova

Pedagogická fakulta

Katedra informačních technologií a technické výchovy

BAKALÁŘSKÁ PRÁCE

Komparace nástrojů pro testování v PHP

Comparison of PHP testing tools

Jakub Frajt

Vedoucí práce:	PhDr. Josef Procházka, Ph.D.
Studijní program:	B7507 – Specializace v pedagogice
Studijní obor:	Informační technologie se zaměřením na vzdělávání

2018

Odevzdáním této bakalářské práce na téma Komparace nástrojů pro testování v PHP potvrzuji, že jsem ji vypracoval pod vedením vedoucího práce samostatně za použití v práci uvedených pramenů a literatury. Dále potvrzuji, že tato práce nebyla využita k získání jiného nebo stejného titulu.

V Praze 5. prosince 2018

Rád bych poděkoval panu PhDr. Josefu Procházkovi, Ph.D. za jeho cenné rady při vedení mé bakalářské práce. Dále děkuji rodině a přátelům za pochopení, že jsem jim při tvorbě této práce nemohl věnovat tolik času.

ABSTRAKT

Předmětem bakalářské práce je vysvětlení problematiky testování webových aplikací v PHP. Teoretická část práce je věnována úvodu do testování webových aplikací, jednotlivých typů testů a agilních metodik, které používají testy v rámci procesu vývoje.

Praktická část práce se zabývá vybranými nástroji pro testování v PHP. Jejich instalací, konfigurací a možnostmi, které nástroje nabízí. Součástí rozboru jednotlivých nástrojů jsou praktické ukázky jednotlivých testů. Podle stanovených kritérií jsou vyhodnoceny důležité vlastnosti a možnosti testovacích nástrojů.

KLÍČOVÁ SLOVA

Testování, testování webových aplikací, agilní metodiky, PHP, testovací nástroje

ABSTRACT

The subject of the bachelor thesis is an explanation of problems of web application testing in PHP. The theoretical part is dedicated to the introduction to testing of a web application, individual types of tests and agile methodologies that use tests in the development process.

Practical part deals with selected tools for PHP testing. Their installation, configurations, and features offered by the tools. Practical examples of tests for each tool are included as part of an analysis. Depending on the criteria, important features and capabilities of test tools are evaluated.

KEYWORDS

Testing, a web application testing, agile methodologies, PHP, testing tools

Obsah

Úvod

Testování v oblasti počítačového softwaru představuje komplexní a velmi důležitou část samotného vývoje. V některých metodikách vývoje se můžeme setkat s přístupem, kdy vývoj software začíná nejprve psaním testů. Je tedy zřejmé, že testování software má zásadní vliv na jeho kvalitu, udržitelnost a běh aplikace. Velkou měrou přispívá ke spolupráci většího počtu vývojářů a poskytuje svým způsobem popis/dokumentaci fungování aplikace. Z tohoto důvodu se s testováním jako takovým setkáme u jednotlivých metodik vývoje, kde představuje nedílnou součást v procesu vývoje aplikace.

Z pohledu uživatele se softwarem setkáme v různých podobách na nepřeberném množství zařízení či online služeb. Každé zařízení/služba představuje oddělený ekosystém na základě použitého operačního systému nebo prostředí (klientské aplikace, webové prohlížeče). Vznikají různá specifika nejen vývoje pro vybranou platformu, ale také testování.

Vývoj webových aplikací představuje komplexní soubor technologií a nástrojů, které se každým dnem neustále vyvíjí. Díky tomu se možnosti aplikací na webu posouvají stále kupředu. Dnes už nemůžeme říci, že by webový vývoj byl odlehčenou „verzí vývoje" software. Už dávno jsou pryč doby, kdy internetu vládly jednoduché HTML stránky s převážně textovým obsahem bez multimédií. Setkáváme se dennodenně s webovými aplikacemi různého typu, které v lecčem překonávají nativní aplikace operačního systému. V některých případech dokonce tyto nativní aplikace nahrazují, díky technologiím jako například Electron[1]. Jedná se o platformu, kde se často pracuje s více programovacími jazyky zároveň. U klasické webové aplikace typicky spolupráce skriptovacího jazyka u klienta v prohlížeči a jiného jazyka na serveru. To klade vysoké nároky jak na vývoj aplikace, tak na nástroje a způsoby testování.

Cílem této práce je prostudovat agilní metodiky, které pracují s testováním v rámci procesu tvorby aplikace, zjištění možností aktuálně dostupných nástrojů pro testování v oblasti webového vývoje v jazyce PHP a vytvoření praktických ukázek testů v těchto nástrojích.

[1] *Electron - Build cross platform desktop apps with JavaScript, HTML, and CSS.* [online]. San Francisco [cit. 2018-11-24]. Dostupné z: https://electronjs.org/

Skriptovací jazyk PHP jsem vybral na základě jeho rozšířenosti[2] a univerzálnímu způsobu použití pro téměř jakýkoliv typ aplikace běžící na straně serveru. Zároveň se také často vyučuje na školách jako zástupce pro programování na webu. Cílem práce je poukázat na možnosti aktuálně dostupných nástrojů pro testování v jazyce PHP. Součástí je komparace nástrojů s přihlédnutím k možnostem vybrané knihovny/nástroje, způsob použití, možnosti jednotlivých testů a aplikace na reálném příkladu. Jednotlivé ukázky testů budou vytvořeny v PHP verze 7.2. s využitím volně dostupných knihoven a nástrojů.

[2] ZAPPONI, Carlo. *GitHut - Programming Languages and GitHub* [online]. 2014 [cit. 2018-09-24]. Dostupné z: https://githut.info/

1 Základní pojmy

Klient

Nejčastěji webový prohlížeč, např. Firefox nebo Google Chrome, vytváří požadavek na server. Úkolem klienta je správné zobrazení souborů nebo zpracování a vykreslení webové stránky (HTML, CSS a JavaScript), které server vrací (Peterka, 2006).

Webový server

Z pohledu hardware je to počítač připojený k internetu. Software je tvořen minimálně HTTP serverem tak, aby byl server schopen přijímat a vyřizovat HTTP požadavky (Mills a kol., 2014).

Na webovém serveru běží webová aplikace, například v PHP. Server tedy na základě požadavků klienta provede potřebné operace, spustí aplikaci a výsledek vrátí jako HTTP odpověď klientovi.

Na vztahu klienta a serveru je charakteristické zejména to, že aktivita je vždy na straně klienta, zatímco server je vysloveně pasivní: své služby sám nikomu nevnucuje, ale pouze pasivně čeká, až o ně nějaký klient požádá (Peterka, 1997).

Mock

Mock je speciální objekt používaný v testech. Je to v podstatě náhrada reálného objektu se závislostmi. Mock objekt simuluje chování reálného objektu bez napojení na reálné závislosti, a navíc umožňuje stanovovat tzv. expectations (Bergmann, 2017).

Můžeme tedy kontrolovat, zda konkrétní metoda byla spuštěna. Typickým příkladem je mock připojení k databázi nebo klient pro vytvoření requestu na službu třetí strany.

System Under Test (SUT)

Termín označuje, co v rámci testu testujeme. Jedná se tedy o nativní implementaci, která je v rámci testu spuštěna a ověřujeme její správné chování a funkčnost. Může se jednat o třídu, metodu nebo celou aplikaci na základě typu testu (Gerard Meszaros, 2007).

2 Testování

Cílem testování a návrhu testů, jako součást kvality kódu, by mělo být zaměření na prevenci chyb (Beizer, 1990). Testování nám tedy umožní předejít potenciálním problémům před vydáním aplikace na produkci nebo při doplnění/rozšíření stávající logiky aplikace.

Testy můžeme odhalit chybějící funkcionalitu nebo nesprávné chování oproti specifikaci aplikace. *Účelem testování je ukázat, že program má chyby* (Beizer, 1990).

Díky jednotlivým typům testů a jejich automatizaci je možné na každé úrovni provádět kontrolu její funkčnosti aplikace pomocí jednotkových, integračních nebo uživatelských testů. Tím můžeme zajistit nejen prevenci chyb, nesprávného zobrazení (UI), ale také výkonových problémů samotné aplikace.

2.1 Specifika testování na webu

Testování na webu představuje samostatnou kapitolu pro tvorbu a spouštění testů. V rámci webové aplikace musí vývojář počítat s úplně jiným přístupem uživatelů. U desktopových aplikací zpravidla určuje hranice operační systém. Na webu je situace jiná a musíme počítat s různými přístupy a několika typy klientů. V případě webové aplikace je velká část na vývojářích aplikace.

Do webové aplikace mohu jako uživatel přijít přes URL adresu na libovolnou část aplikace, aniž bych musel předem projít například přes vstupní formulář či úvodní sekci. Přístup na specifickou část aplikace může být z vyhledávače nebo například přes odkaz ze sociální sítě. Tyto „možnosti" uživatele cestovat po webové aplikaci téměř libovolně, způsobují větší množství uživatelských vstupů, které vývojář aplikace musí v rámci své aplikace nastavit a validovat.

Webovou aplikaci uživatel neinstaluje, může tak přijít na webovou stránku nebo aplikaci s libovolným webovým prohlížečem. Je záležitostí aplikace řešit kompatibilitu a rozdílné chování prohlížečů, které může mít vliv i na vstupní data poslané na server, například skrze webový formulář.

Příklady vstupů a komunikace webové aplikace:

- Přístup uživatele, na již konkrétní sekci webu.

- Webové formuláře, data z nich může klient poslat dvojím způsobem skrze POST nebo GET požadavek, kde součástí mohou být i multimediální soubory.
- Komunikace s API třetích stran.
- Komunikace s databází, např. s MySQL.

Testy jsou jedním z nástrojů a postupů jak proměnlivost webového prostředí a širokou škálu vstupů udržet pod kontrolou.

2.2 Cíle testování a chyby aplikací

Za cíl testování můžeme považovat:

- Stabilní aplikace bez chyb.
- Dodržení funkční specifikace aplikace.
- Transparentní a jednoduchý kód aplikace, dokumentace kódu.
- Udržitelnost kódu s větším počtem vývojářů.

Je důležité si před samotným testováním uvědomit pro jakou aplikaci chceme tvořit testy. V ideálním prostředí bychom měli mít pokryto testy 100 % kódu aplikace. V reálném světě je to dost často nereálné, kvůli ekonomickým a časovým nákladům. Jiným způsobem budeme přistupovat k softwaru, který je velmi důležitý a zachraňuje životy, a jinak k softwaru, který slouží pro evidenci filmové hudby.

Jak uvádí Mirtes (2007), je dobré si rozdělit aplikaci na hlavní logiku aplikace, která je z celé aplikace nejdůležitější a na zbytek aplikace, kde řešíme výpis informací do šablon apod. Pro hlavní logiku aplikace, kde probíhají výpočty nebo probíhá na základě dat validace stavů, by měly v aplikaci existovat udržované testy, které kontrolují hlavní část aplikace, tj. business logiku aplikace. U zbylé části kódu podle časových a ekonomických možností se můžeme obracet na další mechanismy, které alespoň částečně zastoupí funkci testů, jako je například statická analýza kódu.

2.2.1 Chyby aplikace a jejich typy

Testy mohou řešit několik druhů chyb. Mezi ty nejčastější patří:

- Špatný výsledek početních operací/algoritmů.
- Kontrola kroků jednotlivého algoritmu, zda se provedlo všechno, co se mělo provést.

- Kontrola chybových stavů a výjimek.
- Nesprávné argumenty metod a funkcí. Od PHP verze 7 se situace značně zjednodušuje při použití typových kontrol v definici funkce/metody.
- Nesprávná nebo chybějící data, typicky od klienta v případě webového API.
- Syntaktické chyby a překlepy.

3 Typy testů

3.1 Jednotkové testy

První zmínky o použití jednotkových testů (Unit tests) se objevují kolem roku 1970. Kent Beck přišel s konceptem testování pomocí jednotkových testů v rámci programování v jazyce Smalltalk. Postupně se koncept rozšířil do ostatních programovací jazyků. U všech dnes používaných programovacích jazyků jako Java, PHP nebo JavaScript existují nástroje (frameworks), které relativně snadnou cestou umožňují v daném jazyce psát jednotkové testy.

Jak už název napovídá, jedná se o testy, které jsou určeny pro testování nějaké ucelené jednotky kódu, jenž jsme schopni odděleně a nezávisle spustit/zkompilovat. Ve většině případů se jedná o třídu nebo funkci, která dělá právě jednu věc. Jednotkové testy nám umožňují automatizovaně testovat dílčí část kódu, správnou i nesprávnou funkčnost kódu, práce s chybovými stavy a výjimkami.

Jednotkové testy nezjistí, jestli celý dům bude stát. Testujeme jednotlivé cihly, maltu, tvárnice, tedy základní stavební prvky, a ověřujeme, jestli fungují tak, jak od nich očekáváme (Malý, 2011).

3.1.1 Vlastnosti jednotkového testu

Test by měl splňovat základní vlastnosti, které umožní plnou automatizaci, tj. automatizované a vícenásobné spuštění. Díky těmto vlastnostem má test odpovídající hodnotu o tom, zda kód funguje tak, jak je očekáváno.

- Test by mělo být možné nezávisle spustit opakovaně. Neměl by být například závislý na aktuálním datu. Při dalším spuštění by test neprošel. Je to vlastnost, která nám umožňuje testy spustit například pomocí integračního serveru (CI server).
- Test by měl být oddělený od ostatních testů. Znamená to tedy, že by mělo být možné spustit test zcela samostatně, nezávisle na ostatních testech v repozitáři.
- Test by měl pokrývat nejen správné, ale i nesprávné chování kódu. Například nevalidní vstupní data nebo správné vrácení výjimky.

- Test měl pokrýt kompletní funkcionalitu kódu, která je přístupná z venku, veřejné (public) metody a funkce. Již z principu není možné testovat privátní metody a funkce, které nejsme schopni napřímo spustit/vykonat.

3.1.2 Jednotkově testovatelný kód

Pro vytvoření jednotkového testu je zapotřebí, aby kód, který chceme testovat, splňoval základní požadavky pro jeho testovatelnost. Svým způsobem nás testy mohou donutit psát jednoduchý kód, který je správně kompozičně rozdělen a využívá předávání závislostí z venku třídy či funkce.

V tomto ohledu nám značně pomůže návrhový vzor Dependecy Injection. Jedná se o doporučení/vzor, jakým způsobem by se měli v rámci aplikace předávat externí závislosti v kódu. Díky použití tohoto vzoru je možné v rámci jednotkových testů použít tzv. mock objekty pro externí závislosti a tím mít test bez vedlejších efektů a plně pod kontrolou.

Příklad takové závislosti může být databázové připojení. Například kód, který řeší práva uživatelů. Pro ukázku budeme mít metodu, která přidá uživateli další roli v systému. Například můžeme chtít přidat roli redaktora apod. V případě, že budeme psát jednotkový test, který testuje správné přiřazení další role k uživateli, tak nechceme, aby každé spuštění testu vytvořilo připojení do databáze. Jednak bychom porušili pravidlo, že test je zcela izolovaný a nezávislý, jednak by test trval déle než v případě použití mock objektů. Správný přístup v tomto případě je tedy vytvoření mock objektu pro připojení k databázi a testování logiky pro udělení oprávnění.

```
class UserRoleManager
{
    /**
     * @var Connection
     */
    private $connection;

    /**
     * @param Connection $connection
     */
    public function __construct(Connection $connection)
    {
```

```
        $this->connection = $connection;
    }

    /**
     * @param User $currentUser
     * @param Role $role
     *
     * @return User
     * @throws UserRoleException
     */
    public function addRoleToUser(User $currentUser, Role $role): User
    {
        // ... logika metody
    }
}
```

Ve výše uvedené ukázce můžeme vidět použití Dependecy Injection. Využívá se předání závislosti na databázi přes konstruktor třídy. Připojení k databázi není vytvořeno uvnitř třídy, ale je to záležitost kódu, který bude pracovat s touto třídou. Následně v jednotkovém testu můžeme využít mock databázového připojení a soustředit se na otestování logiky metody.

Všechny závislosti by měly být předávány z venku aktuálního modulu/třídy. Zajistíme tím nejen testovatelnost, ale také větší transparentnost kódu. Na první pohled je potom zřejmé, jaké náš kód vyžaduje externí závislosti.

3.2 Integrační testy

Integrační testy představují vyšší úroveň testů z pohledu testovaných částí aplikace. Tento typ testu často pracuje už s reálnými závislostmi, jako je databáze nebo služba třetí strany, například API pro získání PSČ města. To je podstatný rozdíl oproti jednotkovým testům, které pro závislosti jednotlivých tříd a funkcí používají mocking. Pokud test vyžaduje pro své spuštění závislosti, jež nemáme v rámci testu plně pod kontrolou, například právě připojení k databázi nebo spolupráce s modulem na jiném serveru, tak v takovém případě test považujeme za integrační. Pokrývá totiž chování několika modulů/systémů.

3.2.1 Vlastnosti integračních testů

Jednotkové testy jsou relativně rychlé, vše běží v paměti. Nejsou totiž používány reálné závislosti, ale používá se mocking externích tříd a modulů. O integračních testech to říci nemůžeme. Kvůli již zmíněným závislostem jsou testy často mnohonásobně pomalejší. Integrační testy totiž simulují reálné chování aplikace, připojení k databázi nebo odeslání požadavku na server. Konfigurace a spuštění integračního testu bývá často komplikovanější než u jednotkového testu. Z tohoto pohledu už o integračních testech nemůžeme říci, že by byly nezávislé. Je zde závislost na konkrétních službách, které musí běžet pro úspěšné provedení testu.

3.2.2 Kompozice integračních testů

Integrační testy je zpravidla těžké udržovat. Spuštění testu může vyvolat v aplikaci různá připojení a komunikaci jednotlivých subsystémů aplikace. Snadno můžeme dojít do stádia, kdy nebudou integrační testy procházet. Z tohoto důvodu by integrační test měl být napsán na konkrétní případ, například pro konkrétní část API, který chceme testovat metodou požadavek/odpověď (request/response). V opačném případě, kdy integrační testy nebudou testovat dílčí sady aplikace, ale budou pokrývat několik oblastí aplikace, se snadno dostaneme do situace, kdy testy nebudou kvůli častým změnám procházet. Jakákoliv menší změna způsobí, že test neprojde. Navyšujeme tím čas na údržbu takových testů.

Představme si situaci, kdy bychom psali integrační testy pro klasický stolní počítač a jeho komponenty. Spolupráce hardware a software je taktéž v popisu práce u integračních testů. Měli bychom se zaměřit na spolupráci úzce spjatých komponent. Mohli bychom test napsat tak, že pokrývá kompletní funkčnost, tedy počítač „běží“. Když takový test odhalí chybu, tak máme informaci o tom, že je něco špatně. Nicméně nemáme žádné vodítko k tomu, který subsystém počítače je nefunkční. Je proto dobré integrační test pro konkrétní subsystém, například řadič a pevný disk, vytvořit zvlášť jako oddělený integrační test. Chybu tak snadno odhalíme a test bude lépe udržovatelný.

3.3 Akceptační testy

Akceptační testy, někdy také nazývané jako „blackbox“ testy, představují testy na nejvyšší úrovni aplikace z pohledu zákazníka. Často představují poslední úroveň testu před tím, než

je aplikace nebo její nová verze vydána pro produkční spuštění. Cílem akceptačního testu není testovat všechny možnosti chování aplikace. Testy pokrývají jednotlivé procesy chování zákazníka a představují kontrolu nad tím hlavním, zda je schopen uživatel aplikaci používat.

Na akceptačních testech se nemusí již podílet přímo jen vývojáři. Pro tvorbu testu není zapotřebí znát technické detaily, jak aplikace uvnitř funguje. Z tohoto důvodu se na tvorbě testu často podílí přímo testeři, kteří nemusí znát přímo programovací jazyk, ve kterém je aplikace napsaná. Stačí jim znát syntaxi a jednotlivé metody pro tvorbu testovacího scénáře, pomocí kterého popisují průchod aplikací.

3.3.1 Vlastnosti akceptačního testu

Akceptační test je tvořen tzv. uživatelskými scénáři. Scénář popisuje situaci v aplikaci a konkrétní kroky k jejímu provedení. Příkladem scénáře akceptačního testu může být soubor kroků potřebných pro přidání zboží do košíku v internetovém obchodě. Scénář definuje jednotlivé kroky, které vedou z pohledu uživatele k přidání zboží do košíku. Tím se ověřuje dílčí funkcionalita aplikace.

Scénář může mít také podobu sepsaného postupu, myšlenkové mapy. Další možností je využití automatizace pomocí jednoduchých příkazů nebo jednoduchého skriptovacího jazyka. Ve výše zmíněném příkladu s košíkem v internetovém obchodě by scénář konkrétně definoval postup uživatele a čím musí projít pro vložení zboží do košíku. Pokud se akce podle scénáře nezdařila, test končí chybou.

Spuštění a provedení automatizovaných akceptačních testů nepatří mezi ty nejrychlejší. Podobně jako u integračních testů potřebujeme kompletně připravenou aplikaci pro spuštění testu. U webových aplikací simulujeme procházení uživatele jednotlivými částmi, tudíž pracujeme také s reálnými závislostmi. Akceptační testy proto většinou spadají do kategorie nejpomalejších testů.

3.3.2 Kompozice a použití akceptačního testu

Obdobně jako u integračních testů je vhodné akceptační testy dělit na jednotlivé funkční celky aplikace, které chceme testovat. Odděleně se bude testovat proces registrace nového

uživatele a proces přidání zboží do košíku. Výsledkem je lepší udržitelnost testu a čitelnost z pohledu diagnostiky chyb.

Pokud jsou akceptační testy správně začleněny do procesu celého vývoje, tak mohou značně pomoci při simulaci a ověření konkrétních postupů v aplikaci nejen vývojářům ale hlavně testerům, kteří nemusí základní scénáře testovat ručně.

4 Agilní metodiky vývoje s použitým testováním

Při vývoji software se můžeme setkat s různými metodikami vývoje. Je důležité si uvědomit, že tyto metodiky závisí na týmu. Představují model, jakým způsobem ve vývojovém týmu pracovat a stanovují všechny důležité procesy, které vývojový tým řeší. Ne každá metodika je vhodná pro každý tým. Záleží také na produktu, který se vyvíjí. Zvolený model by měl odpovídat potřebám vývoje a pomáhat vývojářům doručovat kvalitní software podložený testy. V neposlední řadě musíme brát v potaz také velikost týmu. Z pohledu procesu vývoje bude tým o velikosti několika desítek vývojářů fungovat jinak než tým např. o pěti vývojářích.

Agilní metodiky představují populární model vývoje. Staly se symbolem malých firem, kde je důraz na zkrácení vývojového cyklu a rychlejší vydání aplikace na produkci. Následně se dostaly do větších firem a dovolím si tvrdit, že dnes jsou nepsaným standardem. Proti například standardnímu modelu Waterfall, který byl jeden z nejznámějších a nejpoužívanějších tradičních modelů vývoje přináší agilní metodiky vícenásobné krátké iterace, důraz na osobní komunikaci v rámci týmů a testování software v průběhu celého vývojového cyklu.

Waterfall spočíval v „přetékání“ práce z jedné fáze do druhé. V případě začátku další fáze byla předchozí fáze brána jako hotová. Což sebou přináší neflexibilitu, protože jednotlivé fáze modelu představovaly velké časové úseky. Zároveň také některé fáze model neumožňuje jednoduše opakovat. V případě chyby v rámci testování, která poukáže na problém v návrhu aplikace, vzniká velkým problém a v podstatě je potřeba začít velkou část vývoje znovu od začátku na základě změny návrhu (Page, Johnston a Rollinson, 2009).

Agilní vývoj je odpovědí na tuto neflexibilitu. Pružnost a možnost kdykoliv změnit směr patří do základních znaků agilních metodik. Cílem těchto metodik je vývoj v rámci malých změn. Je to způsob, jak lépe software udržovat a rychleji tyto malé změny nasazovat na produkci (Page, Johnston a Rollinson, 2009).

Je důležité si uvědomit, že agilní vývoj představuje určitou filozofii vývoje. Často je brán jako něco, co dokáže zvýšit produktivitu a s tímto pohledem je na vývojový tým agilní proces aplikován. Ke zvýšení produktivity samozřejmě může dojít. Nicméně hlavním benefitem

zavedení agilního vývoje je schopnost software vydávat mnohem častěji než pomoci tradičních metodik a pružněji reagovat na změny. Jedná se o změnu práce, nikoliv zrychlení vývoje software (Shore a Warden, 2008).

Agilní metodiky představují jednotlivé procesy, které podporují agilní filozofii. Tyto procesy jsou v agile složeny z jednotlivých elementů/praktik (Shore a Warden, 2008). Příkladem těchto praktik může být používání tzv. coding standards (sjednocený způsob psaní kódu mezi vývojáři) nebo využití verzovacího systému. Tyto praktiky většinou nepředstavují nic nového. Byly používány již v tradičních modelech vývoje. V agilních metodikách jsou jen vhodně uspořádány.

V následujících kapitolách se budu věnovat metodikám, které přímo ve svých praktikách pracují s testováním. Některé metodiky jako SCRUM nebo Kanban nebudou uvedeny, jelikož přímo nepracují v rámci svých definic s konkrétními praktikami, ale věnují se spíše práci v týmu a jednotlivým procesům organizace a komunikace v týmu. Je tedy až na samotném týmu, jakou praktiku pro testování zvolí v rámci práce např. ve SCRUM.

4.1 Extrémní programování (Extreme Programming, XP)

Název je odvozen z toho, co metodika dělá se standardními postupy. Využívá známé principy a postupy softwarového vývoje, které dotahuje do extrémů.

XP počítá s vývojovými týmy o velikosti dvou až deseti programátorů. Jedná se tedy o metodiku pro menší týmy/firmy, kde se i v průběhu vývoje mění zadání.

Tým, který pracuje pomocí této metodiky aplikuje všechny fáze vývoje současně. To znamená, že se neustále pracuje na specifikaci/analýze, návrhu, programování a testování podle potřeby a priorit funkcí vyvíjených v jednotlivých iteracích.

4.1.1 Praktiky používané v XP

Kent Beck (2002), hlavní autor XP, definuje tzv. praktiky, které představují základní elementy vývoje v rámci XP.

Test Driven Development (TDD)

TDD, vývoj řízený testy, představuje postup, kdy vývojář tvoří kód nejprve tím, že píše test. Na postupně vytvořený test píše aplikační kód tak, aby test procházel bez chyb. Je to opačný

přístup, než se běžně používá. Výhodou je kompletní pokrytí kódu jednotkovými testy. V případě, že je tento způsob vývoje v týmu dodržován, se nestane, že by vývojář nevytvořil pro kód test. Nicméně to vyžaduje jistou změnu myšlení v programování a chvíli trvá, než si tento postup člověk osvojí. Na druhou stranu vývojář je takto automaticky „nucen" psát testovatelný kód.

Refaktorizace

XP je silně zaměřeno na výsledné funkce aplikace v každé iteraci. Často je zaměřeno na funkce, které jsou viditelné pro zákazníka a přináší nějakou hodnotu. Z toho důvodu je velmi důležitá správná architektura aplikace, které umožňuje flexibilně logiku upravovat a přidávat nové funkce podle požadavků. Refaktorizace je proces kontinuálního zlepšování kódu a architektury aplikace. Patří sem taktéž optimalizace kódu.

Párové programování

Produkční kód aplikace je tvořen pomocí párového programování. Na kódu se podílejí dva vývojáři zároveň. Výsledkem je kvalitnější aplikační kód a testy, které viděl alespoň jeden další vývojář. Takže v rámci vývoje probíhá rovnou kontrola kódu (code review).

Jednoduchý design aplikace

Vývojáři se snaží psát pochopitelný a jednoduchý kód, který je testovatelný a snadno rozšiřovatelný. Po celou dobu vývoje je kladen důraz na jednoduchost.

Coding Standard

Představuje formátování kódu. Záleží na domluvě v rámci týmu na preferovaných konstruktech jazyka. Cílem by měl být jednotný kód napříč repozitářem aplikace. To pomáhá jak v čitelnosti kódu, tak v hledání chyb a začlenění nového vývojáře do procesu vývoje aplikace.

Udržitelný vývoj

XP pracuje se standardní 40 hodinovou pracovní dobou týdně. Občasný přesčas v počtu jednotek hodin je v pořádku. Pokud ale pracujeme po dlouhou dobu např. 60 hodin týdně, tak není něco v pořádku. XP vychází z toho, že pokud člověk pracuje 60 hodin týdně, tak není odpočatý a nemůže být svěží, kreativní a svědomitý. To samé platí o víkendech, kdy by se člověk neměl věnovat práci. Zmiňuje taktéž důležitou hodnotu možnosti si vzít

dovolenou. Kent Beck (2002) přímo uvádí: „Pokud by se jednalo o mou firmu, trval bych na tom, aby si lidé každý rok vybírali dvoutýdenní dovolenou, a aby měli ještě jeden či dva týdny k dispozici na kratší přestávky.“

Continuous Integration

Představuje nejčastěji server, tzv. CI server, který má za úkol dle stanovených pravidel sestavit aplikaci. V případě webových aplikací s PHP sestavit aplikaci tak, aby bylo možné ji nasadit na server, často přes nějaký Ansible skript nebo sestavením nového Docker image. Výhodou pravidelných sestavení aplikace je spouštění testů a možnost zjistit, zda s integrací nové funkce není problém v jiné části aplikace. Díky tomu je možné mít testovací verzi aplikace k dispozici i několikrát za den.

Metafora

Tým v rámci XP pracuje se stejnou vizí fungování systému. V XP tomuto říkáme metafora. Pomáhá mezi jednotlivými členy týmy pochopit fungování aplikace a tým tak využívá stejné označení k identifikaci technických částí systémů. Používá tedy stejné termíny a označení pro jednotlivé funkcionality systému.

Společné vlastnictví

Není stanoveno, kdo musí pracovat na konkrétní části a kdo do některých částí nemůže zasahovat. Všichni vývojáři mohou pracovat na celém kódu aplikace. Není zde jednoznačně určena role, konkrétní vývojář, který by byl zodpovědný za dílčí funkcionalitu aplikace. Což s sebou přináší výhody z pohledu komunikace. Na druhou stranu se také můžeme dostat do situace, kdy nikdo z týmu nezná, jak nějaká specifická část aplikace funguje. Tyto situace se XP snaží eliminovat praktikami jako je párové programování a tvorba testů.

Tým jako celek

Tým táhne za jeden provaz a v zásadě jsou všichni dostupní v rámci kanceláře dané společnosti. Sedí vedle sebe jak programátoři, tak testeři a produktoví manažeři společně s vedením. Což podporuje komunikace a rychlejší zpětnou vazbu uvnitř týmu.

Plánovací hra

Probíhá takzvané plánování dodávky a plánování iterace. V rámci plánování dodávky určuje zákazník požadavky a vývojáři stanoví časové odhady a náročnost. Na základě časových

odhadů je vytvořen hrubý plán projektu, který se postupně zpřesňuje v jednotlivých iteracích.

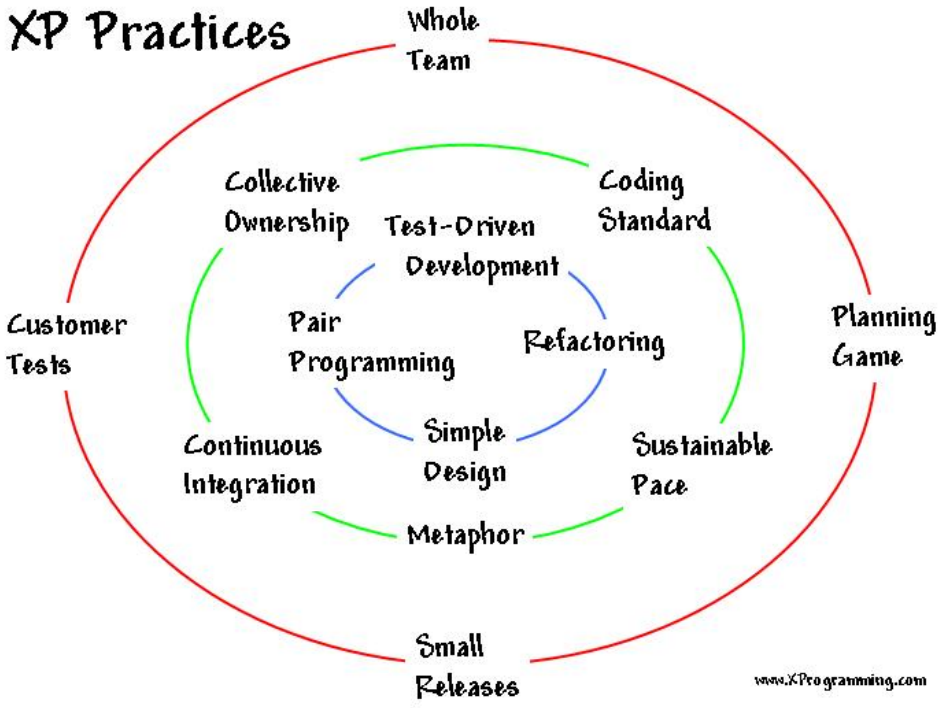
U plánování iterace se řeší plán pro konkrétní např. týdenní iteraci. Na základě toho, co se stihlo a nestihlo, tým zařadí do další iterace nové úkoly.

Malé verze

Tým se soustředí na uvedení první jednoduché verze na trh. Následně uvolňuje malé přírůstkové verze v rámci iteračních cyklů.

Testování

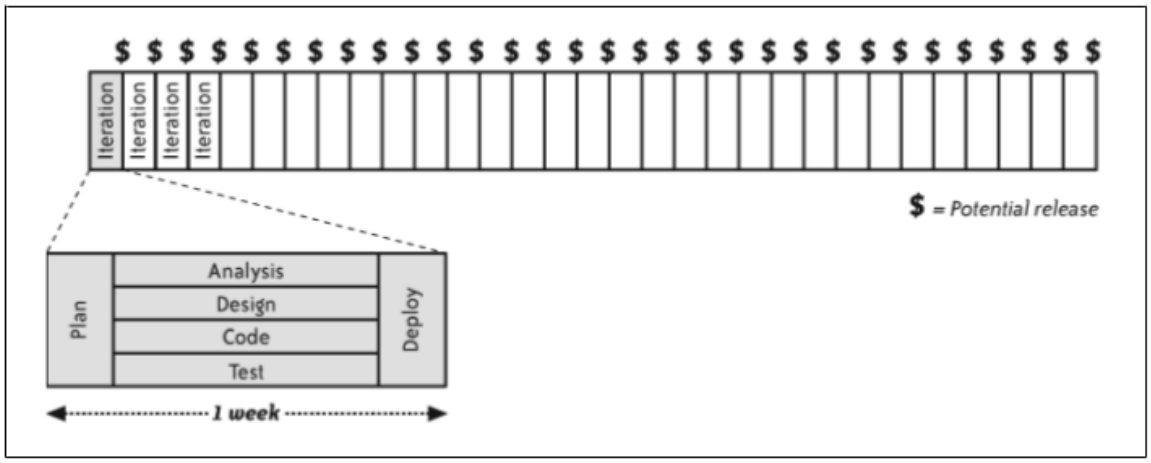
Programátoři mají test pokrytý jednotkovými testy, které musí probíhat bez chyb, aby mohl vývoj pokračovat. Zákazníci specifikují akceptační testy. Většinou je úkolem testera, aby na základě popisu testu vytvořil akceptační test.



Obrázek 1 Principy a postupy využívané v rámci XP. Zdroj: Jeffries, Ron. XP Practices. In: RonJeffries.com [online]. Jeffries, 2011b [cit. 2018-11-24]. Dostupné z: https://ronjeffries.com/xprog/what-is-extreme-programming/circles.jpg

4.1.2 Cyklus vývoje v rámci XP

XP staví na velmi malých iterační cyklech obvykle v délce jednoho až čtyř týdnů, v rámci kterých probíhají všechny standardní činnosti vývoje naráz. Vše je řízeno podle priorit. Přednostně se odbavují funkce s vyšší prioritou. Díky tomu se podle potřeby v daném iteračním cyklu více vyvíjí kód, testuje nebo pracuje na návrhu aplikace. Podle aktuálního stavu a požadavků se přizpůsobí časová kapacita jednotlivých činností. To představuje zásadní rozdíl proti standardním metodikám vývoje jako je Waterfall, kde se dílčí fáze vývoje dělají postupně a je jasně definované pořadí. XP nikde přímo nedefinuje, kolik času by mělo být věnováno jednotlivým fázím vývoje. Je to čistě věc týmu a vedení. Pro XP je cílem v každé iteraci vydat dílčí verzi aplikace. Z toho důvodu se neustále pracuje, prakticky každý den iteračního cyklu, na analýze, návrhu, programování a testování. Tým na konci iterace vytvoří nové sestavení aplikace, které obsahuje novou funkcionalitu pro zákazníka a případné opravy chyb.



Obrázek 2 Znázornění cyklu vývoje v rámci metodiky extrémního programování (Shore a Warden, 2008)

Aplikace metodiky XP neznamená, že by byl tým více produktivní, ale spíše dostává frekventovaně zpětnou vazbu ke každé vydané dílčí verzi aplikace s vybraným souborem nové funkcionality. Tým je schopen mnohem rychleji reagovat na chyby a v rámci krátkého časového úseku je opravit. Zároveň je možné díky zpětné vazbě zákazníka upravit i případný plán, když se ukáže, že bude lepší se vydat jinou cestou. Zamezí se tím vývoji funkcí, které se nebudou nakonec používat, ukázaly se například jako nerelevantní z business hlediska.

Výsledkem je práce týmu na menších nových funkcích nebo jednotlivých dílčích částech větší funkcionality. Těmto malým změnám, na kterých tým v rámci iterace pracuje, se říká *stories* (Shore a Warden, 2008).

4.2 Vývoj řízený vlastnostmi (Feature Driven Development, FDD)

Mezi formálnější představitele agilních metodik, které mezi své praktiky řadí formu testování, patří vývoj řízený vlastnostmi. Za konceptem FDD stojí tři autoři, kteří spolupracovali na velkém bankovním softwarovém projektu. Projektový manažer projektu Jeff De Luca, hlavní architekt aplikace Peter Coad a Stephen Palmer jako manažer vývoje. Metodika byla aplikována na tým o 50 lidech poprvé v roce 1997.

FDD patří mezi agilní metodiky, ačkoli definuje konkrétní procesy vývoje. Mezi hlavní znak patří důraz na modelování aplikace před samotným vývojem. Přichází s novou metodou v UML diagramech, kterou později publikuje Peter Coad v rámci své publikace *Java Modeling in Color* v roce 1999. Je to způsob, kdy jednotlivé části UML diagramu jsou odlišeny různou barvou pro lepší znázornění a pochopení jednotlivých částí domény aplikace (Palmer a Felsing, 2002).

Metodika pracuje, jak je u agilních metodik typické, s iteračními cykly. Zpravidla v délce dvou týdnů. Dbá na kvalitu, kterou se snaží zajistit pomocí praktik jako jednotkové testování nebo coding standards podobně jako v metodice extrémního programování. Výsledkem je frekventované doručení nových verzí aplikace a uchopitelný výstup napříč každým procesem FDD.

Primárním cílem této metodiky je vývoj client-valued features – vlastností a funkcí, které mají nějaký přínos pro uživatele. Na základě toho staví své procesy a použité praktiky.

Využití této metodiky připadá v úvahu většinou u větších týmů, kde je zapotřebí přesně definovaný proces. Typicky pro týmy, které fungovaly na bázi metodiky Waterfall.

4.2.1 Praktiky používané v FDD

Doménové objektové modelování (Domain Object Modeling)

Velký důraz v FDD je kladen na modelování jednotlivých vlastností a funkcí, anglicky *features*, softwarového projektu. Doménový model aplikace představuje vizuální pohled na

jednotlivé části aplikace a umožňuje dekomponovat jednotlivé funkční celky. Výsledkem je koncept integrace patřičných vlastností aplikace. Autoři FDD ve své knize zmiňují techniku „modeling in color", která přináší do UML diagramů barvu. Barva pomáhá k lepší kategorizaci funkčních celků a vizuálně nám dává pohled na úzce související elementy aplikace (Palmer a Felsing, 2002).

Vývoj podle vlastností (Developing by Feature)

Podobně jako i v jiných metodikách FDD pracuje s tím, že pokud bychom vyvíjeli danou vlastnost více jak 2 týdny, tak je zapotřebí ji rozdělit na menší celky. To nám umožní tvořit reálnější časové odhady a možnost implementace v kratším časovém horizontu.

Příkladem vlastnosti/funkce aplikace může být:

- možnost řazení dat, např. výpisu transakcí, podle data, názvu
- historie úprav dokumentu u software pro vytváření dokumentů

Vlastnictví kódu/tříd

FDD používá tzv. individuální vlastnictví kódu/tříd, opačný přístup než například XP. Jako základní element pro nějaký funkční celek se bere v FDD třída. V objektově orientovaných programovacích jazycích třída představuje základní strukturu pro zapouzdření kódu, který tvoří ucelenou funkční jednotku. V zásadě jsou jednotlivým vývojářům přiřazeny dílčí části domény aplikace, za kterou jsou z pohledu kódu zodpovědní. Mají na starosti údržbu a kontrolu nad přidáním nových funkcionalit. Můžeme tam tedy zařadit konzistenci, výkon a koncepční integritu třídy. Mezi výhody takto rozdělené zodpovědnosti mezi vývojáři patří:

- Schopnost vysvětlit do detailu, jak daná část aplikace funguje. Vývojář se stává expertem na příslušnou část systému.
- Vývojář vlastník (owner) je schopen mnohonásobně rychleji zakomponovat nějaké vylepšení do kódu. Zná kód do detailu oproti ostatním vývojářům.

Z výše uvedených výhod vyplývají také nedostatky. Úzkým hrdlem zde může být komunikace. Vývojáři se musí v případě překryvu funkcionality mezi více subsystémů mezi sebou domluvit. Je zde také možnost odchodu jednoho z vývojářů a tým tak přichází o znalost části systému.

Opakem je kolektivní vlastnictví v rámci XP. Nevýhodou je samozřejmě nejednoznačná zodpovědnost a rozhodovací funkce, které se XP snaží řešit svými praktikami jako je párové programování.

Týmy podle vyvíjených vlastností (Feature Teams)

Podobně jako u konkrétních dílčích částí aplikace/tříd, tak i u jednotlivých vlastností aplikace FDD používá přerozdělení zodpovědnosti. Vznikají týmy, které sdružuje tým lídr, který má ve svém týmu vývojáře, kteří jsou vlastníky kódu pro vybranou vlastnost aplikace. Typicky tým lídr zastřešuje několik částí aplikace.

Inspekce (Inspections)

Inspekce představují způsob kontroly návrhu aplikace a samotného kódu. Není zde přímo v reálném čase kontrola od druhého vývojáře jako v případě XP. Nicméně probíhá kontrola návrhu aplikace před samotnou implementací a taktéž kontrola kódu po implementaci. Pro kód platí, že před inspekcí kódu by měly být napsány jednotkové testy pro implementaci.

Pravidelné sestavení aplikace (Regular builds)

Jako u ostatních agilní metodik, tak i FDD definuje používání pravidelných sestavení aplikace na definované časové bázi dle možností a velikosti projektu/týmu.

Řízení konfigurací (Configuration Management)

FDD zdůrazňuje, že nejenom pro samotný kód by měla být udržována historie, ale i pro dokumenty a popisy související s danou verzí, jednotlivá zadání funkcí pro konkrétní verze apod. Víceméně se dá říci, že pro jakékoliv dokumenty týkající se projektu, ke kterým by v budoucnu byla potřeba se vrátit.

Reporting/viditelnost výsledků (Reporting/Visibility od Results)

FDD taktéž počítá s častým reportingem o stavu vývoje tak, aby vedoucí společnosti a odpovědní manažeři dokázali reagovat na rychlost vývoje a řídit tak priority, určovat další směr.

FDD používá procentuální vyjádření o stavu implementace vybrané funkce, což představuje základ pro sestavení shrnujícího reportu. Zavádí práci se šesti milníky, které pokrývají návrh

a implementaci nových vlastností. Procenta jsou přerozdělena na jednotlivé milníky. Podle dosažených milníků potom odpovídá reportovaný stav vedení a vedoucímu vývojáři.

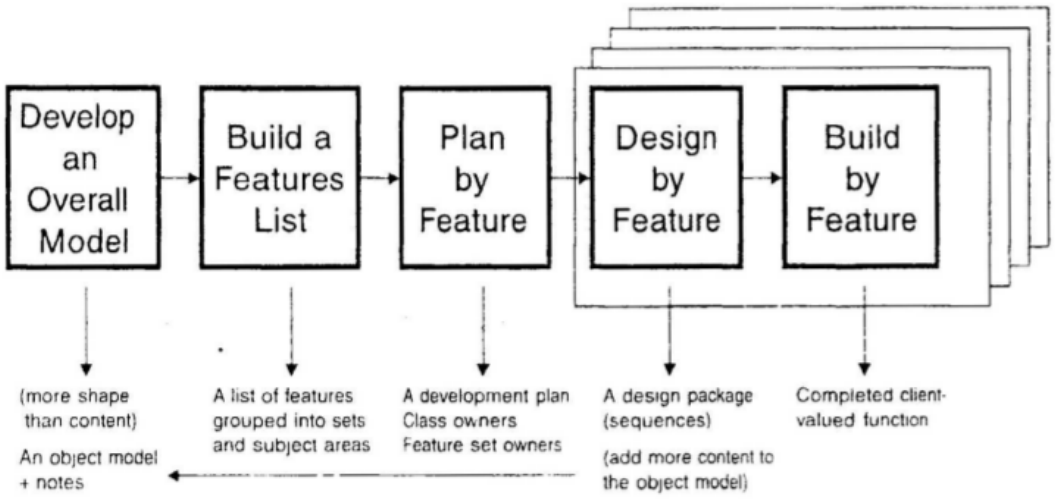
Proces návrhu			Proces vývoje		
Procházení doménového modelu, analýza	Návrh konkrétních tříd/metod	Inspekce návrhu	Tvorba kódu	Inspekce kódu, testy	Vytvoření nového sestavení
1 %	40 %	3 %	45 %	10 %	1 %

Tabulka 1 Příklad jednotlivých podílů činností pro reporting v rámci FDD (Palmer a Felsing, 2002)

Výsledný aktuální stav implementace je stanoven součtem již hotových milníků, nikoliv rozpracovaných. Stanovení vah jednotlivých milníků by měly odpovídat reálným předpokladům daného týmu.

4.2.2 Cyklus vývoje v rámci FDD

FDD popisuje proces vývoje pomocí pěti procesů ve kterých specifikuje potřebné kroky.



Obrázek 3 Grafické znázornění pěti procesů FDD s jejich výstupem (Palmer a Felsing, 2002)

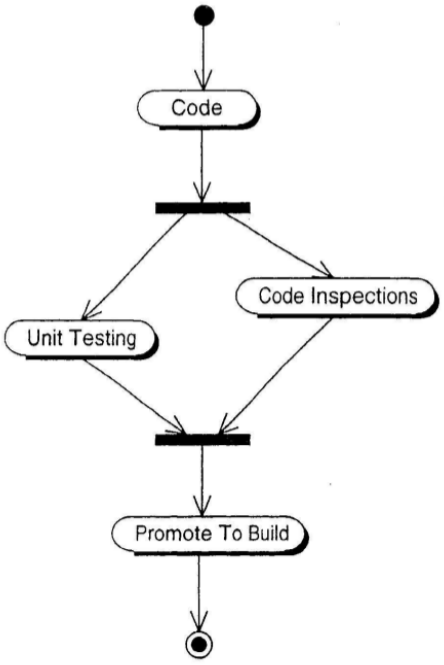
V rámci prvního procesu má tým za úkol sestavit doménový model, nad kterým se výsledná aplikace bude tvořit. Pod vedením hlavního architekta aplikace vytvoří diagram

doménového modelu, ve kterém jsou zohledněny představy a požadavky uživatelů. Tento diagram slouží následně jako dokumentace a prostředek pro sestavení konkrétních funkcí a vlastností aplikace, které se budou v iteračních cyklech vyvíjet. Výstupem toho procesu je kostra aplikace, diagram jednotlivých tříd. Na obrázku výše vidíme také znázornění zpětné vazby do doménového modelu. V případě změn či nových funkcí by měl být model udržován aktuální.

Následně tým vývojářů v druhém procesu vytvoří seznam vlastností aplikace. Vstupem pro vytvoření tohoto listu vlastností je již zmíněný doménový model a veškeré další důležité dokumenty, které jsou potřeba pro vývoj této funkce, např. nějaké dokumentace software třetích stran nebo funkční specifikace. Výsledný soubor vlastností koresponduje s potřebami zákazníků i s business požadavky společnosti.

Třetí proces řeší prioritu jednotlivých vlastností. Výstupem je pořadí, ve kterém se jednotlivé vlastnosti budou vyvíjet, a konkrétní tým lídři, kteří budou zodpovědní za vývoj těchto vlastností.

Ve čtvrtém procesu tým lídr určí, kterých tříd se nové vlastnosti budou týkat. Vybraní vývojáři řeší návrh nových vlastností, tedy návrh architektury, a jakým způsobem budou nové vlastnosti zapracovány. Po této dekompozici nových vlastností probíhá inspekce, kontrola a ladění návrhu.



Obrázek 4 Build by Feature proces FDD (Palmer a Felsing, 2002)

Po odsouhlasení návrhu následuje pátý proces, samotný vývoj nových vlastností, kde vlastníci tříd naprogramují nové vlastnosti a funkce podle domluveného návrhu. Probíhá vytvoření jednotkových testů a inspekce kódu.

Po úspěšné inspekci vytvořeného kódu a odsouhlasení hlavního programátora se vytváří nové sestavení aplikace.

Poslední dva procesy se iterativně opakují pro každou další novou vlastnost, která se podle plánu bude vyvíjet.

4.3 Test Driven Development (TDD)

TDD se zaměřuje čistě na tvorbu kódu v kombinaci s testy. Nezabývá se procesy ani plánováním, ale čistě tvorbou kódu pomocí testů. Používá opačný přístup pro psaní testů než je většinou u vývojářů a používaných metodik zvykem. Základem je tvorba testu ještě před existencí implementace. Vyplývá z toho, že bez testu nevytvoříme kód. Výstupem je tedy testovatelný kód, pro který existují jednotkové testy. Výhodou plynoucí z použití TDD je možnost kód neustále spouštět. Kód jsme díky testům schopni kdykoliv spustit. Odpadá tedy spousta problémů a dodatečných scénářů testování, kdy vývojář tvoří dlouho kód logiky,

kterou nemá možnost jednoduše spustit a řeší chyby až na konci při celkovém spuštění, a nikoliv v průběhu tvorby.

4.3.1 Praktiky a tipy v TDD

Test list

Před samotným testováním je dobré si sestavit seznam funkcí, které budeme chtít testovat. Lze ho pokládat za seznam vlastností a funkcí, které třída bude mít.

Assert First

V rámci TDD testu většinou řešíme několik problémů, jako závislosti, co se bude volat a jakým způsobem se to bude jmenovat. Za dobrou techniku lze považovat napsání konkrétního výsledku, který bude předmětem kontroly před tvorbou implementace. Následně se dalšími kroky snažíme dostat k tomuto výsledku assertions.

Fake It

Po prvotním testu, který selže je dobré vytvořit implementaci, která vrací výsledek staticky, například jako konstantu, abychom si ověřili správnou funkčnost testu. Nutí nás to tedy psát co nejjednodušší implementaci. Například pokud bychom psali funkci *sum()* pomocí TDD a využili falešné implementace, tak by byl postup následující:

```
function testSum(): void {
    $this->assertEquals(8, sum(1, 7));
}
```

Implementace:

```
function sum(int $a, int $b): int {
    return 8;
}
```

Používání falešné implementace abychom se dostali do zeleného stavu, kdy testy prochází, má dvě výhody:

- Psychologickou – mít test, který prochází je určitě lepší než mít test, který je červený a neprochází. Víme, že test je napsaný správně a prochází. Můžeme se následně

věnovat refaktorizaci implementace. Pokud uděláme chybu, tak máme jistotu, že test neprojde a budeme o tom vědět.

- Rozsah testu a implementace – jako vývojáři řešíme v tu chvíli právě jeden problém. Soustředíme se na refaktorizaci jedné dílčí části.

Triangulace

Podobně jako má význam triangulace v matematice, kde představuje způsob na zjišťování souřadnic vzdáleností, tak v rámci TDD nám může pomoci při hledání vhodné implementace. Pokud bychom chtěli psát test pro funkci *sum()* pomocí triangulace, tak by to vypadalo následovně:

```
function testSum(): void {
    $this->assertEquals(8, sum(1, 7));
}
```

Implementace:

```
function sum(int $a, int $b): int {
    return 8;
}
```

V prvním kroku bychom využili falešné implementace. Následně bychom doplnili další assert, který způsobí, že test skončí chybou:

```
function testSum() {
    $this->assertEqual(8, sum(1, 7));
    $this->assertEqual(3, sum(0, 3));
}
```

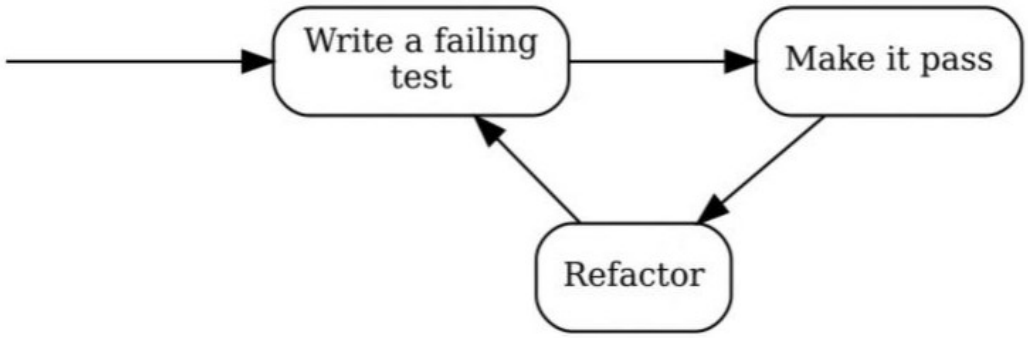
Díky tomu, že jsme vytvořili další porovnání, tak test nebude procházet. Naše implementace v tuto chvíli není správná a je potřeba ji zobecnit. Jak uvádí Ken Beck ve své knize Test Driven Development, triangulace má specifické použití. Mohli bychom se snadno dostat do smyčky, kdy provádíme falešné implementace, které vrací správná data. Teoreticky bychom mohli rozšířit falešnou implementaci tak, aby splňovala i druhé porovnání.

Zřejmá implementace (Obvious implementation)

Pokud tvoříme jednoduchou logiku, tak není nutné využít technik jako Fake It nebo Triangulation. Pokud víme, jak danou logiku hned napsat, tak bychom ji měli napsat. Test nám potvrdí, zda je naše myšlenka správná. Obvykle se to nevyplácí u složitější logiky, kde hned na první pokus nemusíme dojít k správnému řešení.

4.3.2 Cyklus vývoje v TDD

Často se můžeme setkat s termínem „red-greed refactor", což je v podstatě označení pro proces/postup vývoje v TDD. Vývojář napíše test pro konkrétní situaci, například pro zavolání metody s určitými argumenty, a očekává definovaný výstup. Test v tomto případě skončí jako neúspěšný, protože samotná implementace ještě není vytvořena. V tu chvíli nastává čas, kdy vývojář přechází z testu do implementační části. Doplní logiku tak, aby existující test byl úspěšný a procházel. Je důležité zmínit, že vývojář by v tomto bodě měl doplnit minimum logiky tak, aby test procházel a zároveň logika nepřesáhla šíři aktuálního testu. Následuje refaktorizace kódu, kde odstraníme nežádoucí duplicity a učiníme logiku obecnou, odpovídající testům. Tento cyklus se opakuje tak dlouho, dokud není popsána testy kompletní logika metody nebo celé třídy.



Obrázek 5 TDD cyklus vývoje (Gałęzowski Grzegorz, 2016)

TDD začíná s tím, že test neprojde. Je to z toho důvodu, abychom předešli chybám před implementací a nesprávným nastavením testovací frameworku. Každý test musí alespoň jednou skončit chybou, abychom měli důkaz o jeho správném spuštění a funkčnosti. Pracuje se zde také s psychologickým efektem. Termín "red-green refactor" je odvozen od chování testovacích frameworků, které v případě, že test neprochází zobrazí červené upozornění. V opačném případě, kdy test prochází, zobrazí zelené upozornění.

Typicky se při vývoji pomocí TDD používá tzv. test runner. Jedná se o nástroj, který nám pravidelně testy spouští při jakékoliv úpravě implementace nebo testu. Ihned je potom dostupný výsledek testu a můžeme na to reagovat. Většinou lze nastavit toto chování buď přímo přes testovací framework nebo ve vývojovém prostředí, které nám automaticky dokáže v předem stanovených intervalech testy spouštět.

5 Nástroje pro testování v PHP

Výběr jednotlivých nástrojů byl učiněn na základě popularity a aktivity vývoje dle statistik serveru Github[3].

5.1 PHPUnit

PHPUnit je základním, ale velmi komplexním, nástrojem pro tvorbu jednotkových testů. Autorem je Sebastian Bergmann, který tento testovací framework stále vyvíjí a je nejaktivnějším přispěvatelem do kódu podle statistik Githubu.

Aktuálně podporované jsou verze 6 a 7, které pracují již s PHP verze 7 a výše. Pro PHP verze 5 už není PHPUnit udržován. Z podstaty věci to nemá smysl, jelikož PHP 5.6 bude končit bezpečnostní podpora koncem roku 2018.

5.1.1 Instalace a zprovoznění

Instalace PHPUnit je možná buď pomocí přímého stažení PHP archivu, který obsahuje vše potřebné anebo přes Composer. Composer je v PHP ekosystému hlavním nástrojem pro řešení externích závislostí aplikací. Doporučuji instalaci přes Composer, přes který můžeme jednoduše spravovat verze závislostí a nemusíme řešit jejich stahování. Instalaci přes Composer spustíme v kořenové složce projektu následujícím způsobem:

```
composer require --dev phpunit/phpunit ^7
```

V případě, že dodržíme jednoduchou strukturu adresářů, tedy testy budeme ukládat do složky *tests*, tak PHPUnit automaticky rekurzivně nalezne testy a bude je spouštět. Pokud bychom chtěli využít jiné adresářové struktury, tak je zapotřebí PHPUnit nakonfigurovat přes konfigurační XML soubor.

[3] GitHub [online]. San Francisco, 2008 [cit. 2018-11-25]. Dostupné z: https://github.com/

5.1.2 Možnosti nástroje

Testování výsledku metody

Porovnání a kontrola výsledku metody patří mezi základní mechanismy jednotkového testu. PHPUnit poskytuje celkem širokou škálu podpůrných metod, které slouží pro testování výstupu metody.

Mezi ty základní můžeme zařadit:

- **assertEquals** – kontroluje přesnou shodu výsledku očekávané hodnoty. Pro desetinná čísla umožňuje stanovit přesnost s jakou bude čísla porovnávat. Používá se také pro porovnání objektů na základě hodnot properties objektu.
- **assertTrue/assertFalse** – slouží pro testování boolean hodnoty na výstupu. V testu tedy očekáváme hodnotu *true* nebo *false*.
- **assertCount** – využijeme v případě, že potřebujeme kontrolovat například počet zpracovaných záznamů.
- **assertContains** – lze využít, jak už je z názvu patrné, pro testování výskytu prvku ve výsledku metody. Metodu můžeme využít pro práci s řetězci i polem.
- **assertNull** – porovnání, zda je explicitně výstupem hodnota *null*.
- **assertSame** – představuje porovnání na základě hodnoty a datového typu.
- **assertEmpty** – kontrola, zda je výsledek např. prázdné pole.
- **assertInstanceOf** – můžeme testovat, zda je na výstupu správná instance třídy
- **assertLessThan/assertGreaterThan** – kontrola, zda je hodnota v definovaném intervalu.
- **assertJsonStringEqualsJsonFile/assertXmlStringEqualsXmlString** – můžeme rovněž přímo pracovat s datovým formátem JSON nebo XML.

Testování výjimek

Pokud testovaná logika řeší i chybové stavy pomocí výjimek, tak v PHPUnit můžeme testovat:

- Typ výjimky, kterou logika za dané situace vrátí.
- Kód výjimky.
- Zprávu, která je součástí výjimky, nebo její určitou část.

Ukázka všech dostupných kontrol pro výjimky:

```
class ExceptionsTest extends TestCase
{
    public function testException(): void
    {
        $this->expectException(\RuntimeException::class);
        $this->expectExceptionCode(20);
        $this->expectExceptionMessage('a message for an exception.');
        $this->expectExceptionMessageRegExp('/message/');
        (new Foo())->throwRuntimeException();
    }
}
```

Testování textového výstupu

V některých případech potřebujeme otestovat i textový výstup metod. Může se jednat o CLI skript nebo skript, který budeme spouštět plánovaně v časových intervalech pomocí systémového služby (Cron). Pro tyto případy PHPUnit umožňuje kontrolovat textový výstup skriptu:

```
public function testOutputString(): void
{
    // testovani vystupu na presnou shodu
    $this->expectOutputString('expected string');

    // nebo pomoci regularniho vyrazu
    $this->expectOutputRegex('/expected/');

    $foo = new Foo();
    $foo->print();
}
```

Data Providers

PHPUnit pomocí tzv. Data Providers umožňuje lépe pracovat s testovacími soubory dat. Typickým příkladem je test, který obsahuje několik kontrol, assertions. V takovém případě je jedna z možností testovat každý výsledek pomocí metody zvlášť:

```
public function testWithMultipleAssertions(): void
{
    $calc = new Calc();

    $this->assertSame(2, $calc->sum(1, 1));
    $this->assertSame(7, $calc->sum(6, 1));
    $this->assertSame(12, $calc->sum(8, 4));
}
```

Alternativou je využití odkazu přes anotaci na vytvořený Data Provider. Díky tomu je možné mít pouze jednu kontrolu v testu:

```
/**
 * @dataProvider validSumDataProvider
 */
public function testWithMultipleAssertions(): void
{
    $calc = new Calc();
    $this->assertSame(12, $calc->sum(8, 4));
}

public function validSumDataProvider(): array
{
    return [
        [1, 1, 2],
        [6, 1, 7],
        [8, 4, 12],
   ];
}
```

Výhodou je rychlejší a jednodušší zápis v případě, kdy chceme v testu kontrolovat několik různých variant vstupů. Dále také oddělení testovacích dat od samotného kódu testu.

Používání anotací

Dodatečnou možností konfigurace testů jsou anotace. Představují rozšíření či alternativu ke konfiguraci.

Příkladem je anotace *@dataProvider*, která slouží pro odkaz na vytvořený Data Provider. Pokud je u testu použita *@test* anotace, tak nemusí metoda testu začínat předponou *test*. Je tak možné použít libovolný název metody testu.

Test doubles

Test Doubles představují obecný název pro simulování chování reálných objektů – mock objects. PHPUnit nabízí dvě metody pro vytvoření mock objektů – *createMock()* a *getMockBuilder()*. Na základě předané třídy nebo rozhraní (interface) jako argumentu do jedné z těchto metod, PHPUnit vytvoří objekt, který se vůči okolnímu kódu jeví jako nativní implementace.

Ačkoliv PHPUnit nevyužívá přímo v názvu metod rozlišení pro typy testovacích objektů, tak při vytváření testovacích objektů můžeme narazit na rozdílná chování a možnosti. Setkáme se zde s následujícími typy testovacích objektů:

- **Dummy objekt** – v případě využití metody *createMock()* jsou všechny metody originální implementace třídy nahrazeny prázdnou implementací s návratovou hodnotou *null*. Zároveň nejsou volány metody *__construct()* a *__clone()*. Tento typ testovací objektu nazýváme typicky dummy objektem. Často se tyto objekty vytvářejí pro splnění závislostí testované třídy jako argument, ale v testu nejsou použity přímo (Fowler, 2004).
- **Stub objekt** – vytvoření objektu s využitím nahrazení nativní implementace testovacím objektem a možností nadefinovat návratovou hodnotu metody objektu představuje tzv. stubbing (Bergmann, 2018).
- **Mock objekt** – rozšiřuje v podstatě stub objekt. Pracujeme zde navíc s informací, zda byla metoda testovacího objektu zavolána (Marshall, 2004). Nastavujeme tzv. expectations. Jsme tedy schopni kontrolovat, kolikrát byla metoda volána a s jakými argumenty například.

Často se pracuje jen s názvem mock, jak mezi vývojáři, tak v různých publikacích.

Mocking Abstract class, Traits

Stejně jako můžeme vytvářet mock objekty ze standardní třídy, tak můžeme vytvořit mock z abstraktní třídy nebo traits:

```
public function testWithMockingTraitsOrAbstractClass(): void
{
    $mock = $this->getMockForTrait(FooTrait::class);
    //...
    $mock = $this->getMockForAbstractClass(AbstractFoo::class);
    //...
}
```

5.1.3 Praktická ukázka použití

V první ukázce testujeme implementaci validátoru, který může sloužit pro validace PIN kódu zadaného přímo uživatelem na straně klienta ve formuláři, nebo při validaci na úrovni nějakého API.

Test využívá tzv. fixtures metod, konkrétně metodu *setUp()*, která je spuštěna před každým dílčím testem a umožňuje si před každým testem připravit data a proměnné. V tomto případě se vždy vytváří nová instance validátoru.

Kód testu pro PIN validátor:

```
class PinValidatorTest extends TestCase
{
   /**
    * @var PinValidator
    */
    private $validator;

    public function setUp()
    {
        $this->validator = new PinValidator();
    }
```

Každý test v PHPUnit vychází ze základní třídy *TestCase*, která obsahuje potřebné metody pro testování. Dále vidíme použití metody *setUp()*. Díky tomu se před každým testem vytvoří nová instance validátoru.

V další části testujeme správné/validní kombinace pro PIN. V tomto případě by validátor neměl zahlásit chybu. Pro testování několika kombinací se využívá data provider:

```
/**
 * @dataProvider validPinProvider
 */
public function testValidPin(string $value): void
{
    $this->assertEmpty($this->validator->validate($value));
}
public function validPinProvider(): array
{
    return [
        ['4479'],
        ['4513'],
        ['9761'],
        ['0036'],
    ];
}
```

Stejný postup testu je aplikován i pro nevalidní kombinace, kdy testujeme nepovolené znaky:

```
/**
 * @dataProvider invalidPinProvider
 */
public function testInvalidPin(string $value): void
{
    $this->assertGreaterThan(0, $this->validator->validate($value));
}

public function invalidPinProvider(): array
{
    return [
        ['abab'],
        ['12as'],
        ['AAAA'],
        ['126?'],
    ];
}
```

V dalším testu kontrolujeme explicitně kombinace, které jsou zakázané, a validátor by měl vrátit konkrétní validační zprávu:

```
/**
 * @dataProvider invalidCombinationsProvider
 */
public function testInvalidCombinations(string $value): void
{
    $this->assertEquals(
        ['This combination is not allowed.'],
        $this->validator->validate($value)
    );
}
public function invalidCombinationsProvider(): array
{
    return [
        ['1234'],
        ['4321'],
    ];
}
```

Poslední dva testy jsou vyčleněny na kontrolu validačních chybových zpráv pro případ, kdy PIN nesplňuje minimální počet znaků, anebo je součástí PIN kombinace znak, který není číslem:

```
public function testInvalidLength(): void
{
    $this->assertEquals(
        ['PIN must contains exactly four numbers.'],
        $this->validator->validate('23')
    );
}
public function testNotNumericPin(): void
{
    $this->assertEquals(
        ['PIN must contains only numbers'],
        $this->validator->validate('12a5')
    );
}
```

Druhou ukázkou je třída, která slouží pro odesílání pravidelného newsletteru aktivním uživatelům. Na ukázce je demonstrován mocking závislostí, které jsou předávány pomocí

Dependecy Injection přes konstruktor. Můžeme tak vytvořit stub objekt přímo pro repository, které slouží pro získání aktivních uživatelů. Následně mock *MailerInterface*, pro odeslání e-mailu.

V prvním testu je tedy vytvářen stub objekt pro *UsersRepository*. Nastavujeme výsledek metody, která v reálné implementaci vrací aktivní uživatele. V tomto případě je potřeba nasimulovat situaci, kdy nejsou žádní uživatelé aktivní. Tohoto stavu je v testu docíleno tím, že se vrací prázdné pole. Neodesílá se tedy žádný e-mail. Mock objekt má tak nastavené patřičné expectations:

```
class NewsletterSenderTest extends TestCase
{
    public function testSendEmailWhenWeDontHaveActiveUsers(): void
    {
        $usersRepositoryStub =
            $this->createMock(UsersRepository::class);
        $usersRepositoryStub
            ->method('getActiveUsers')
            ->willReturn([]);

        $mailerMock = $this->createMock(MailerInterface::class);
        $mailerMock
            ->expects($this->never())
            ->method('send');

        $newsletterSender = new NewsletterSender(
            $usersRepositoryStub,
            $mailerMock
        );
        $newsletterSender->sendNewsletterToActiveUsers();
    }
```

V druhém testu je stub objekt komplexnější. Vrací tři objekty aktivních uživatelů. Testujeme tedy přesný opak. Metoda *send()* na mock objektu *mailer* se musí zavolat celkem třikrát. Tímto způsobem jsem schopni ověřit, že správně dojde k odeslání e-mailu pro tyto tři uživatele.

```
public function testSendEmail(): void
{
    $usersRepositoryStub = $this->createMock(UsersRepository::class);

    $usersRepositoryStub
        ->method('getActiveUsers')
        ->willReturn([
            $this->createUserWithEmail('jimmy@example.com'),
            $this->createUserWithEmail('mathew@example.com'),
            $this->createUserWithEmail('tony@example.com'),
        ]);

    $mailerMock = $this->createMock(MailerInterface::class);
    $mailerMock
        ->expects($this->exactly(3))
        ->method('send');

    $newsletterSender = new NewsletterSender(
        $usersRepositoryStub,
        $mailerMock
    );
    $newsletterSender->sendNewsletterToActiveUsers();
}

private function createUserWithEmail(string $email): User
{
    return new User(1, 'test', 'test', $email, true);
}
```

5.2 Mockery

Mockery je testovací framework zaměřený na mock objekty. V podstatě se jedná o nadstavbu pro již existující framework typu PHPUnit. Za frameworkem stojí Pádraic Brady a více jak stovka dalších přispěvatelů na Githubu.

Cílem je poskytnout jednoduché a komplexní API pro práci s mock objekty, které zjednoduší čitelnost testů a umožní flexibilně mock objekty vytvářet. Rozšiřuje také soubor testovacích objektů o částečné mock a spies objekty (Brady a kol., 2017).

5.2.1 Instalace a zprovoznění

Jak již bylo uvedeno výše, tak Mockery se zabývá pouze prací s mock objekty. Pro použití v testech je potřeba mít nainstalovaný a zprovozněný například PHPUnit, který poskytuje v tomto případě základ, nad kterým Mockery staví.

Mockery lze nainstalovat standardním způsobem přes Composer následujícím způsobem:

```
composer require --dev mockery/mockery
```

Pro integraci je zapotřebí v testech manuálně pomocí PHPUnit metody *tearDown()*, která se spustí po každém testu, zavolat Mockery metodu *close()*. Tím je řešen reset mock objektů a kontejneru, který interně Mockery používá.

```
public function tearDown() {
    \Mockery::close();
}
```

Mockery nabízí také alternativu pomocí dědičnosti. V testech se potom vychází z třídy *MockeryTestCase* namísto standardní PHPUnit třídy *TestCase*. Nemusíme tak vždy přidávat ručně metodu *tearDown()*.

Nic dalšího není potřeba speciálně nastavovat. Mockery je pomocí standardního autoload systému dostupné pod globálním namespace *\Mockery*. Pro zkrácení zápisu je možné využít aliasu při importu například následujícím způsobem:

```
use \Mockery as m;
```

5.2.2 Možnosti nástroje

Mockery na rozdíl od PHPUnit nabízí jednotný postup jak pro vytvoření standardního mock objektu, tak i pro stub nebo spy objekt. Všechny tyto typy implementují rozhraní *MockeryInterface* a vytvoří se jednoduše přes metodu mock:

```
$mock = m::mock(Foo::class);
```

Stejným způsobem lze vytvořit mock ze standardní třídy, ale také pro interface nebo abstraktní třídu.

Vytváření stub objektů

Pro definici stub objektu lze využít metody *allows()* nebo *shouldReceive()*. Výsledný objekt je stejný, v případě použití jedné z těchto metod. Rozdíl je v možnosti zápisu. U *allows()*

Mockery umožňuje přímo metodu, kterou chceme v stub objektu využít, zapsat v řetězci konfiguračních metod:

```
public function testWithCreatingStub(): void
{
    $stub = m::mock(Foo::class);
    $stub
        ->allows()
        ->testMethod()
        ->andReturns(10);
}
```

V tomto případě je tedy *testMethod()* přímo název metody v implementaci třídy. Případně lze využít i možnosti přidání argumentu. Výsledkem potom metoda, která s určitým argumentem vrací definovaný výsledek.

Pokud nepotřebujeme specifikovat pro jednotlivé metody stub objektu výsledek na základě argumentu, tak můžeme využít definice několik metod najednou pomocí pole. Elegantním způsobem tak definujeme několik metod najednou:

```
public function testWithCreatingMock(): void
{
    $stub = m::mock(Foo::class);
    $stub->allows([
            'testMethod1' => 10,
            'testMethod2' => 20,
        ]);
}
```

Alternativou je využití metody *shouldReceive()*, která byla původně jedinou možností do verze 1.0. Název metody je argumentem:

```
public function testWithCreatingStub(): void
{
    $stub = m::mock(Foo::class);
    $stub
        ->shouldReceive('testMethod')
        ->andReturns(10);
}
```

I v tomto případě je možné specifikovat argument, pomocí Mockery metody *with()*, pro který má stub vracet požadovanou hodnotu.

Vytváření mock objektů

Podobně probíhá vytvoření mock objektu. Na výběr máme z metod *shouldReceive()* a nově také *expects()* pro konfiguraci expectations. V případě použití metody *expects()* lze stejně jako v případě *allows()* využít přímo název metody, kterou chceme kontrolovat, v definici mock objektu. Ve výchozím stavu Mockery nastavuje expectations rovno jedné. To znamená, že nemusíme specifikovat počet volání v případech, kdy bude metoda volána pouze jednou. V opačném případě můžeme explicitně definovat počet volání následujícím způsobem:

```
public function testWithCreatingMock(): void
{
    $mock = m::mock(Foo::class);
    $mock
        ->expects()
        ->testMethod()
        ->twice();
}
```

Spies objekty

Spy objekt je speciálním type mock objektu. U standardního mock objektu musíme nastavit expectations pro vybrané metody před zavoláním SUT. Spy objekt umožňuje vynechat definici expectations. Automaticky totiž zaznamenává všechny interakce s SUT (Brady a kol., 2017).

V ukázce vidíme, že až po zavolání SUT kontrolujeme pomocí metody *shouldHaveReceived()*, zda byla konkrétním metoda volána:

```
public function testWithUsingSpy(): void
{
    $spy = m::mock(Foo::class);
    $bar = new Bar($spy);

    $spy->shouldHaveReceived('testMethod');
}
```

Partial Mocks

Jak už z doslovného překladu vyplývá, jedná se o částečné mock objekty. Mockery umožňuje vytvářet mock, který funguje jako prostředník mezi reálnou implementací (SUT) a mock objektem. Lze tedy specifikovat, pro které metody se využije mocking a pro které se zavolá nativní implementace.

Partial mock lze vytvořit několika způsoby:

- **Runtime partial mock** – metody, které nejsou explicitně nadefinovány v rámci mock objektu, jsou delegovány na nativní implementaci.
- **Generated partial mock** – chování je stejné jako u runtime partial mock objektů. Rozdíl je pouze v tom, že u generovaných se hned na začátku explicitně stanoví, které metody budou součástí mock objektu.
- **Proxied partial mock** – fungují jako proxy a dají se využít pro mocking final tříd. Interně jsou odchytávána volání a přesměrována buď do mock objektu, nebo nativní implementace. Nevýhodou je, že tyto objekty neprojdou přes type hint v PHP, jelikož nevychází přímo z nativní implementace.

Ukázky vytvoření jednotlivých typů partial mocks:

```
// runtime partial mock
$mock = m::mock(Foo::class)->makePartial();
// generated partial mock
$mock = m:mock(Foo::class. '[testMethod]');
// proxy partial mock
$mock = m:mock(new Foo());
```

5.2.3 Praktické ukázka použití

S využitím Mockery testujeme stejnou třídu pro odesílání pravidelného newsletteru aktivním uživatelům jako v případě PHPUnit.

V první testu kontrolujeme, že nedojde k volání metody *send()*. Můžeme využít spy objektu. Nemusíme tak definovat expectations a kontrolu provést až po vykonání SUT.

```
public function testSendEmailWhenWeDontHaveActiveUsers(): void
{
    $usersRepositoryStub = m::mock(UsersRepository::class);
    $usersRepositoryStub
        ->allows()
        ->getActiveUsers()
        ->andReturns([]);

    $mailerSpyMock = m::spy(MailerInterface::class);

    $newsletterSender = new NewsletterSender(
        $usersRepositoryStub,
        $mailerSpyMock
    );
    $newsletterSender->sendNewsletterToActiveUsers();
    $mailerSpyMock->shouldNotHaveReceived('send');
}
```

To samé platí pro scénář, kdy máme nějaké aktivní uživatele. Stejným způsobem bude na konci testu provedena kontrola na konkrétní počet volání metody *send()*:

```
public function testSendEmail(): void
{
    $usersRepositoryStub = m::mock(UsersRepository::class);
    $usersRepositoryStub
        ->allows()
        ->getActiveUsers()
        ->andReturns([
            $this->createUserWithEmail('jimmy@example.com'),
            $this->createUserWithEmail('mathew@example.com'),
            $this->createUserWithEmail('tony@example.com'),
        ]);
```

```
    $mailerSpyMock = m::spy(MailerInterface::class);

    $newsletterSender = new NewsletterSender(
        $usersRepositoryStub,
        $mailerSpyMock
    );
    $newsletterSender->sendNewsletterToActiveUsers();

    $mailerSpyMock->shouldHaveReceived('send')->times(3);
}
```

5.3 Codeception

Jedná se o komplexní testovací framework, který si klade za cíl zpřístupnění tvorby všech základních typů testů pod jedním nástrojem. V rámci Codeception je tedy možné vytvářet a pracovat s těmito typy testů:

- Jednotkové testy
- Integrační testy
- Akceptační a funkcionální testy

Codeception dále poskytuje celou řadu modulů pro dnes používané PHP frameworky, např. pro Laravel, Symfony nebo Yii. Zároveň jsou dostupné i další dodatečné moduly pro práci s různými databázemi (MySQL, PostgreSQL, Oracle) nebo ORM (Doctrine).

Výsledkem je velmi komplexní soubor nástrojů a helper modulů pro práci s různými scénáři a službami.

5.3.1 Instalace a zprovoznění

Samotná instalace je možná dvěma způsoby. První možností je standardní instalace přes Composer:

```
composer require Codeception/Codeception –dev
```

Druhou možností je instalace a používání přes Phar archiv, který použijeme v případě, kdy není možné nebo z nějakého důvodu nechceme použít Composer.

Konfigurace je složitější oproti ostatním nástrojům. Využijeme-li standardní konfigurace, lze následujícím příkazem vygenerovat základní konfigurační soubory a potřebné složky:

```
codecept bootstrap
```

V opačném případě, kdy chceme využít pouze některou součást Codeception, je dobré spustit příkaz pro vygenerování konfigurace pro vybraný typ testu. Např. pro používání jednotkových testů:

```
codecept init unit
```

Jde o zásadní volbu konfigurace. Dílčí konfigurace konkrétních typů testů povoluje přes interaktivní příkaz podrobnější konfiguraci (paths, namespaces).

5.3.2 Možnosti nástroje

K dispozici je několik typů pomocných příkazů, které slouží pro konfiguraci nebo generování základní struktury jednotlivých testů.

```
Available commands:
  bootstrap             Creates default test suites and generates all required files
  build                 Generates base classes for all suites
  clean                 Recursively cleans log and generated code
  console               Launches interactive test console
  dry-run               Prints step-by-step scenario-driven test or a feature
  help                  Displays help for a command
  init                  Creates test suites by a template
  list                  Lists commands
  run                   Runs the test suites
 config
  config:validate       Validates and prints config to screen
 generate
  generate:cept         Generates empty Cept file in suite
  generate:cest         Generates empty Cest file in suite
  generate:environment  Generates empty environment config
  generate:feature      Generates empty feature file in suite
  generate:groupobject  Generates Group subscriber
  generate:helper       Generates new helper
  generate:pageobject   Generates empty PageObject class
  generate:scenarios    Generates text representation for all scenarios
  generate:snapshot     Generates empty Snapshot class
  generate:stepobject   Generates empty StepObject class
  generate:suite        Generates new test suite
  generate:test         Generates empty unit test file in suite
```

Obrázek 6 Jednotlivých možností CLI příkazu codecept

Jednotkové testy

Jednou z hlavních částí Codeception je tvorba jednotkových testů. Interně se využívá standardní PHPUnit. Základní metody pro assertions jsou stejné. Codeception ale přichází

s vlastními metodami pro konfiguraci testu a také s vlastními nástroji pro tvorbu mock objektů. Jako alternativu lze ale použít i Mockery.

Základní strukturu jednotkového testu je možné vygenerovat příkazem:

```
codecept generate:test unit Foo
```

První argument příkazu definuje typ testu a druhý název třídy. Výstupem je vygenerovaná třída pro jednotkový test:

```
class FooTest extends \Codeception\Test\Unit
{
    protected function _before() {}
    protected function _after() {}
    // tests
    public function testSomeFeature() {}
}
```

Tvorba a pojmenovávání testů funguje identicky jako v PHPUnit. Metody *_before()* a *_after()* jsou ekvivalentem pro metody *setUp()* a *tearDown()*.

Test Doubles

S Codeception lze vytvářet stub, mock a dummy objekty. Stub objekt lze vytvářet přes statickou třídu, což na první pohled jasně definuje, že se jedná o stub. Na druhou stranu se jedná stále o alternativu. Ukázka vytvoření jednotlivých typů objektů:

```
public function testCreatingTestObjects(): void
{
   $stub = Stub::make(Foo::class, ['getName' => 'Jan Novák']);
   $dummy = $this->makeEmpty(Foo::class);
   $mock = $this->makeEmpty(Person::class, [
            'getName' => Expected::once(),
            'getAge'  => Expected::atLeastOnce(),
        ]);
}
```

Integrační testy

Pro integrační testy je k dispozici základní modul pro práci s databází zvaný *Db*. Jeho využití přichází v úvahu, když nepoužíváme žádný z podporovaných PHP frameworků nebo

knihoven. Umožňuje kontrolovat, zda se používá připojení do správné databáze a jednotlivé záznamy v tabulkách. Ukázka jednotlivých dostupných helper metod:

```
public function testWithIntegrationHelper(): void
{
    /** @var Db $tester */
    $tester = $this->getModule('Db');

    $tester->seeInDatabase('book', ['id' => 10, 'name' => 'TDD']);
    $tester->dontSeeInDatabase('book', ['id' => 1, 'BDD']);
    $tester->seeNumRecords(10, 'books', ['active' => true]);
    $tester->amConnectedToDatabase('testdb');
    $tester->grabFromDatabase('book', 'id', ['name' => 'TDD']);
}
```

V případě použití podporovaného ORM, například Doctrine, lze ověřovat hodnoty přímo v objektech (repositories).

Akceptační testy

U akceptačních testů dochází ke spouštění a procházení aplikace tak, jak by aplikaci procházel normální návštěvník. Codeception nabízí dvě možnosti, co se týče volby klienta:

- **PhpBrowser** – představuje prohlížeč napsaný v PHP. V zásadě pracuje jen s čistým HTML a není zde žádné grafické rozhraní. Výhodou je relativně rychlé provedení, nespouští se kompletní prohlížeč s grafickým rozhraním a zároveň pro jeho spuštění není potřeba nic navíc kromě PHP s Curl rozšířením. Nevýhodou je, že nepracuje s JavaScriptem, nijak ho nespouští/nezpracovává.
- **WebDriver** – funguje na principu Selenium serveru. Využívá se buď PhantomJS nebo Chrome či Firefox. V tomto případě je možné spouštět i JavaScript. Nevýhodou je pomalejší běh, jelikož je zde kooperace s dalším nástrojem.

Codeception se svým API pro tvorbu akceptačních testů snaží přiblížit srozumitelné tvorbě testu. Scénář testu tedy na první pohled vypadá srozumitelně a i méně technicky zdatnému uživateli nebude dělat problém popsat, co se ve scénáři definuje.

Test je možné psát jako jednoduchý lineární PHP skript nebo pomocí třídy, kde každá metoda představuje jeden test/scénář. Ukázka základních metod pro tvorbu akceptačního testu:

```
$I->amOnPage('/');
$I->see('Headline', 'h2');
$I->canSeeCookie('acceptance');
$I->fillField('name', 'Jan');
$I->selectOption('type', 'new');
$I->click('Next page');
$I->seeInCurrentUrl('/test');
$I->sendAjaxRequest('/get-main-city', [
    'data' => 'Czech Republic'
]);
```

Codeception nabízí dostatek helper metod, které umožňují kontrolovat texty na stránce, číst cookies, dělat Ajax requesty nebo vyplňovat a odesílat formuláře.

5.3.3 Praktická ukázka použití

V první ukázce je jednotkový test pro *NewsletterSender*, který byl již testován přes Mockery a PHPUnit. Kód testu je téměř totožný, lze vidět jen drobné rozdíly, díky rozdílným helper metodám pro mocking.

```
public function testSendEmailWhenWeDontHaveActiveUsers(): void
{
    $usersRepositoryStub = Stub::make(
        UsersRepository::class,
        ['getActiveUsers' => []]
    );

    $mailerMock = $this->makeEmpty(MailerInterface::class, [
        'send' => Expected::never()
    ]);

    $newsletterSender = new NewsletterSender(
        $usersRepositoryStub,
        $mailerMock);
```

```
    $newsletterSender->sendNewsletterToActiveUsers();
}
```

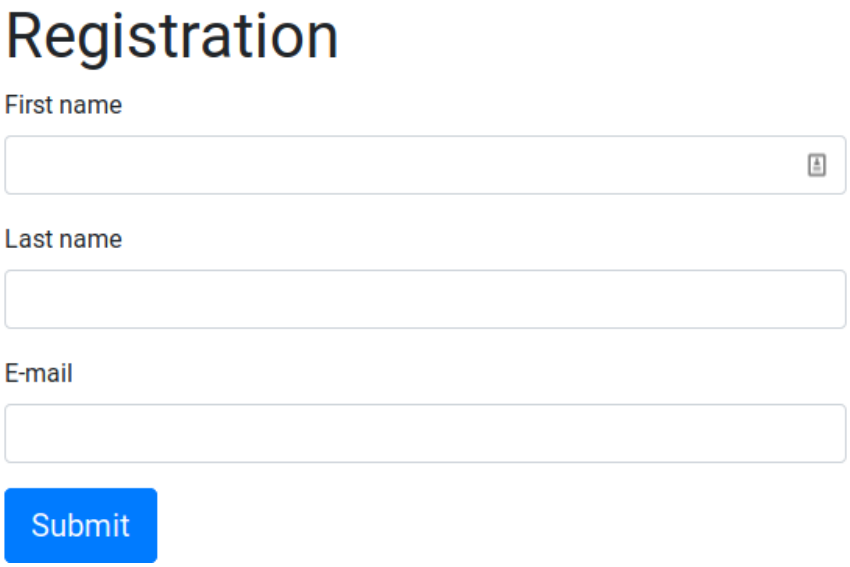
Codeception má relativně dobře připravené metody pro vytváření mock a stub objektů tak, že lze vše nastavit v rámci jednoho řádku. U mock objektů se využívá klasické expectations. V základu neobsahuje Codeception spy objekty.

V druhé ukázce je ukázka tvorby integračního testu s pomocí *Db* helperu. Testuje logiku *UsersRepository*, konkrétně metody *save()*, která má za úkol uložit správně data do databáze.

V první části vytváříme objekt *UserData*, který potom následně předáváme do repository pro uložení do databáze. Přes helper metodu *seeInDatabase()* probíhá kontrola, zda se v tabulce *users* nachází patřičná data:

```
public function testSaveNewNotActiveUser(): void
{
   $userData = new UserData();

   $userData
        ->setFirstName('Jan')
        ->setLastName('Novák')
        ->setEmail('novak@example.com');

    $usersRepository = new UsersRepository($this->getDbalConnection());
    $usersRepository->save($userData);

    $tester = $this->getModule('Db');

    $tester->seeInDatabase('users', [
        'first_name' => 'Jan',
        'last_name'  => 'Novak',
        'created'    => date('Y-m-d'),
        'active'     => 0
    ]);
}
```

Následující ukázka ukazuje propojení celého procesu do kompletní registrace a otestování pomocí akceptačního testu. Uživatel na stránce s registračním formulářem vyplňuje základní údaje a následně jsou data uloženy do databáze. Po úspěšné registraci je zobrazena informační hláška.



Obrázek 7 Registrační formulář, který vyplňujeme v rámci akceptačního testu

Cílem akceptačního testu je v tomto případě ověřit průchod formulářem z pohledu uživatele a zkontrolovat možné validační chyby v případě, kdy uživatel nevyplní všechny údaje.

Akceptační testy, které jsou tvořeny formou třídu, musí obsahovat suffix *Cest*.

V prvním testovacím scénáři test pokrývá standardní průchod registračním procesem, kdy uživatel vyplní všechny údaje správně. Kontroluje se, zda je na stránce patřičný nadpis první úrovně a probíhá vyplnění a odeslání formuláře. Po odeslání formuláře v testu zkontrolujeme i správnou informační hlášku o dokončení registrace:

```
class RegistrationProcessCest {

    public function registrationProcessSuccess(AcceptanceTester $I)
    {
        $I->amOnPage('/');
        $I->see('Registration', 'h1');
        $I->fillField('firstname', 'Jan');
        $I->fillField('lastname', 'Novák');
```

```
        $I->fillField('email', 'novak@example.com');
        $I->click('Submit');
        $I->see('Registration successfully completed.');
    }
```

Ve druhém a třetím scénáři probíhá kontrola chybových stavů validace, kdy uživatel nevyplní požadované údaje nebo vyplní nevalidní e-mail:

```
public function registrationWithEmptyForm(AcceptanceTester $I)
{
    $I->amOnPage('/');
    $I->see('Registration', 'h1');
    $I->click('Submit');
    $I->see('E-mail cannot be empty.');
    $I->see('First name cannot be empty.');
    $I->see('Last name cannot be empty.');
}

public function registrationWithInvalidEmail(AcceptanceTester $I)
{
    $I->amOnPage('/');
    $I->see('Registration', 'h1');
    $I->fillField('firstname', 'Jan');
    $I->fillField('lastname', 'Novák');
    $I->fillField('email', 'novak.com');
    $I->click('Submit');
    $I->see('E-mail is not valid.');
}
```

6 Instalace a spuštění ukázek

Kompletní kód testů s odpovídající implementací je k dispozici na také Githubu:

https://github.com/jakub-frajt/comparison-of-PHP-testing-tools

Pro zprovoznění a spuštění testů je zapotřebí standardní linuxová distribuce typu Debian či Ubuntu. Jediným dodatečným softwarovým požadavkem je nainstalovaný Docker a Docker Compose společně s Git.

Nejdříve je zapotřebí si vyklonovat z repozitáře samotný kód ukázek:

```
git clone https://github.com/jakub-frajt/comparison-of-PHP-testing-tools
```

Následně spustíme instalaci přes bash skript prostřednictvím terminálového klienta v root adresáři projektu:

```
./setup.sh
```

Skript pomocí Composeru nainstaluje externí závislosti projektu a následně pomocí Dockeru vytvoří image pro PHP. Výsledkem je spuštění tří Docker kontejnerů pro Apache server, PHP a MySQL s testovací databází. Součástí je také klientská aplikace Adminer pro prohlížení dat v databázi.

Nyní je možné spouštět testy pro konkrétní testovací framework pomocí bash skriptu *run_tests.sh*. Argumentem je název nástroje – *phpunit*, *mockery* nebo *codeception*:

```
./run_tests.sh phpunit
```

7 Hodnocení nástrojů

7.1 Stanovení kritérií

Kritéria vycházející z praktického použití:

- **Konfigurace a dokumentace** – co všechno je potřeba nastavit a jaký je stav dokumentace.
- **Podporované typy testů** – na co se nástroj primárně orientuje, jaké možnosti z pohledu typů testů nabízí.
- **Tvorba testů a jejich čitelnost** – kritérium zohledňující správný přístup k rozlišení jednotlivých typů testovacích objektů (mock, stub, dummy) z pohledu syntaxe a samotné API nástroje pro tvorbu testů.
- **Vhodné použití** – zohlednění typu použití, na jaké projekty je nástroj vhodné použít.

7.2 Vyhodnocení testovacích nástrojů dle kritérií

7.2.1 PHPUnit

Konfigurace a dokumentace

V případě využití standardního chování, kdy spouští testy v definované adresářové struktuře není potřeba nic specifického nastavovat. Případné začlenění do již existujícího projektu a využití jiné adresářového struktury lze nastavit v rámci jednoho XML souboru. Výsledkem je velmi rychlá instalace s možností v podstatě ihned tvořit testy bez zdlouhavé konfigurace.

Dokumentace je vcelku přehledná, nicméně některé části ohledně tvorbě mock a stub objektů by mohly být vysvětleny srozumitelněji společně s ukázkami.

Podporované typy testů

PHPUnit se zaměřuje primárně na jednotkové testy. Umožňuje nějaké dodatečné rozšíření ve formě emulace souborového systému, ale primární účelem zůstávají jednotkové testy, případně integrační testy. Dříve podporované databázové testy a možnost tvořit integrační testy již nejsou aktivně vyvíjeny.

Tvorba testů a jejich čitelnost

Syntaxe jednotlivých assertions je přímočará a srozumitelná. Za začátku mohou dělat problémy metody *setUp()* a *tearDown()*, kde nemusí být hned na první pohled jasné, co je jejich účelem. To samé lze říci o tvorbě jednotlivých typů testovacích objektů. Pro začátečníka není úplně vhodné, že syntaxe přímo nepracuje s názvy stub a mock explicitně. Vše je skryto za názvem mock. Patřičné rozdíly se vývojář dozví až při důkladném studiu dokumentace. V některých případech, zejména při práci s mock objekty, by mohlo API umožňovat jednodušší zápis.

Vhodné použití

Tím, že se PHPUnit primárně zaměřuje na jednotkové testy, není jeho využití nikterak omezeno. Vychází z jednotlivých nástrojů jako základní kámen pro tvorbu testů, a to jako přímá či nepřímá součást jiných testovacích nástrojů.

7.2.2 Mockery

Konfigurace a dokumentace

Mockery kromě instalace přes Composer neobsahuje žádnou dodatečnou konfiguraci mimo samotné testy. Pokud nepočítáme instalaci PHPUnit, nad kterým staví. Jedinou podmínkou je použití speciální trait v PHPUnit testech nebo využití dědičnosti. Instalace je tedy velmi jednoduchá.

Dokumentace je přehledná a velmi dobře srozumitelná. Nevýhodou je, že některá části dokumentace jsou na oficiálním Githubu a další část na oficiální stránce dokumentace. Uvedené příklady na Githubu by bylo vhodné mít pro přehlednost uvedené i přímo v dokumentaci.

Podporované typy testů

Mockery je nástrojem, který se zaměřuje na tvorbu testovacích objektů, tedy mock, stub a spies objektů. Jeho využití spadá primárně do jednotkových či integračních testů.

Tvorba testů a jejich čitelnost

Mockery na rozdíl od PHPUnit umožňuje alespoň v rámci alternativního API od verze 1 používat explicitní metody pro práci s mock a stub objekty. Což přináší lepší čitelnost

v testech. Nevýhodou je ale stále možnost použít starší syntaxi, což ze začátku může být matoucí pro začátečníky.

Velké plus představuje možnost používat objekty typu spy. To představuje znatelné ulehčení tvorby testů při tvorbě komplexních větší testů a zlepšuje čitelnost jednotlivých testů.

Vhodné použití

V podstatě ho lze využít všude, kde využíváme již PHPUnit nebo jiný podporovaný testovací framework a je kladen požadavkem na rychlejší psaní testů s jednodušším API. Výhodou je rozšíření portfolia testovacích objektů o spies.

7.2.3 Codeception

Konfigurace a dokumentace

Díky tomu, že Codeception podporuje celou řadu typů základních testů, konfigurace nepatří zrovna k nejjednodušším.

Jednotlivé typy konfigurací je možné vygenerovat přes dostupný příkaz *codecept*. Na druhou stranu je ale potřeba říci, že v případě již existujícího projektu či potřeby jiné adresářové struktury moc tato automatizace nepomáhá. Navíc dokumentace se soustředí jen na několik málo příkladů konfigurace pro standardní adresářovou strukturu a moc dobře nepokrývá specifické konfigurace. Dost často je možná konfigurace různými způsoby a pro vývojáře to může být značně matoucí. U jednotlivých testů by to chtělo doplnit jednotlivé kompletní konfigurace, jak testovací prostředí nastavit.

Podporované typy testů

Velkou výhodou je podpora všech základních typů testů pod jednou střechou. Codeception umožňuje tvořit jednotkové, integrační, akceptační testy. Možností je používat i jen určité typy testů, např. akceptační.

Tvorba testů a jejich čitelnost

U jednotkových testů se setkáváme s další alternativou tvorby testovacích objektů. V některých ohledech se také chová Codeception jinak a nahrazuje standardní PHPUnit metody, nad kterými staví. Pro vývojáře, který zná PHPUnit, to může být nepřehledné.

Co se týče integračních testů, tedy modulu pro práci s databází, a akceptačních testů je syntaxe naprosto v pořádku a velmi srozumitelná i pro testera.

Vhodné použití

Osobně se mi jeví vhodné použít tento nástroj v rámci velkého nového projektu nebo v případě, že chceme využít akceptační testy jako samostatný modul. U stávajícího projektu, který má již nějakou historii a codebase, je na zvážení, zda se vyplatí kompletně přejít. Konfigurace je potom mnohem složitější a může zabrat výrazně více času, pokud někdo z týmu nemá již s Codeception nějakou zkušenost.

Závěr

Teoretická část práce je věnována problematice testování webových aplikací. Jsou zde rozebrány jednotlivé agilní metodiky, které ve svých procesech explicitně definují testování. Součástí rozboru agilních metodik jsou i konkrétní praktiky a postupy.

Praktická část se zabývá vybranými nástroji pro testování a rozborem implementace testů. Z pohledu hodnocených nástrojů se ukazuje, že pro programovací jazyk PHP existují relevantní nástroje pro testování, které umožňují plnohodnotné testování webové aplikace. Jednotlivé API pro tvorbu testů jsou u nástrojů vytvořeny s ohledem na jednoduchost a rychlost tvorby testů. Na základě praktických ukázek se ukazuje, že samotná konfigurace může představovat zásadní kritérium pro výběr a začlenění nástroje do již existující aplikace. Je tedy záležitostí jednotlivých týmů, aby podle typu aplikace a způsobu použití vhodně zvolili testovací nástroj.

Jednotlivé ukázky testů poskytují informativní pohled, jakým způsobem nástroje konfigurovat a použít v reálných případech. Díky tomu může být práce vhodná pro učitele i studenty středních a vysokých škol v oborech zaměřených na informační technologie a programování. Zároveň může sloužit jako publikace pro začínající vývojáře a menší týmy, které s testováním teprve začínají a potřebují základní analýzu jednotlivých nástrojů.

Seznam použitých informačních zdrojů

BECK, Kent. Extrémní programování. Praha: Grada, 2002. Moderní programování. ISBN 80-247-0300-9.

BECK, Kent. Test-driven development: by example. Boston: Addison-Wesley, 2003. ISBN 978-0-321-14653-3.

BEIZER, Boris. Software Testing Techniques. Second Edition. New Delhi: Dreamtech Press, [1990]. Reprint Edition: 2017. ISBN 978-81-7722-260-9.

BERGMANN, Sebastian. PHPUnit Manual. In: PHPUnit Manual - PHPUnit 7.4 Manual [online]. Bergmann, 2017 [cit. 2018-11-16]. Dostupné z: https://phpunit.readthedocs.io/en/7.4/index.html

BODNARCHUK, Michael a kol. Codeception [online]. 2011 [cit. 2018-11-24]. Dostupné z: https://codeception.com/

BRADY, Pádraic a kol. Mockery Docs [online]. Brady, 2016 [cit. 2018-11-18]. Dostupné z: http://docs.mockery.io/en/latest/index.html

FOWLER, Martin. Mocks Aren't Stubs. In: Martin Fowler [online]. Chicago: Martin Fowler, 8 July 2004 [cit. 2018-11-04]. Dostupné z: https://martinfowler.com/articles/mocksArentStubs.html

GAŁĘZOWSKI, Grzegorz. Test-Driven Development: Extensive Tutorial [online]. In: . 2016 [cit. 2018-10-19]. Dostupné z: https://archive.org/details/tdd-ebook/page/n0

GOYAL, Sadhna. Agile Techniques for Project Management and Software Engineering. Munich, 2007. Dostupné také z: http://csis.pace.edu/~marchese/CS616/Agile/FDD/fdd.pdf. Technical University Munich.

JEFFRIES, Ron. What is Extreme Programming?. In: RonJeffries.com [online]. Jeffries, 2011a [cit. 2018-11-24]. Dostupné z: https://ronjeffries.com/xprog/what-is-extreme-programming/

JEFFRIES, Ron. XP Practices. In: RonJeffries.com [online]. Jeffries, 2011b [cit. 2018-11-24]. Dostupné z: https://ronjeffries.com/xprog/what-is-extreme-programming/circles.jpg

MALÝ, Martin. Ještě k testování. In: Zdroják [online]. Praha: Devel.cz Labs, 2011, 14. 3. 2011 [cit. 2018-10-20]. Dostupné z: https://www.zdrojak.cz/clanky/jeste-k-testovani/

MARSHALL, Dave. Mocks Aren't Stubs, Fakes, Dummies or Spies. In: YouTube [online]. London: SymfonyLive London, 2014, 24 Nov 2014 [cit. 2018-11-04]. Dostupné z: https://www.youtube.com/watch?v=6_r3AzRg1HM

MESZAROS, Gerard. XUnit test patterns: refactoring test code. Upper Saddle River, NJ: Addison-Wesley, 2007. ISBN 978-0-13-149505-0.

MILLS, Chris a kol. What is a web server?. In: MDN Web Docs [online]. Mozilla Foundation, 2014, 16 Jul 2014 [cit. 2018-11-16]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server

MIRTES, Ondřej. Usnadněte si práci silně typovaným kódem a statickou analýzou!. In: SlidesLive [online]. Praha: SlidesLive, 2017, 1. 4. 2017 [cit. 2018-09-14]. Dostupné z: https://slideslive.com/38900570/usnadnete-si-praci-silne-typovanym-kodem-a-statickou-analyzou

OTADUY, I. a O. DIAZ. User acceptance testing for Agile-developed web-based applications: Empowering customers through wikis and mind maps. JOURNAL OF SYSTEMS AND SOFTWARE [online]. 2017, 133, 212-229 [cit. 2018-10-10]. DOI: 10.1016/j.jss.2017.01.002. ISSN 01641212.

PAGE, Alan, Ken JOHNSTON a Bj ROLLISON. Jak testuje software Microsoft. Brno: Computer Press, 2009. ISBN 978-80-251-2869-5.

PALMER, Stephen R a John M FELSING. A practical guide to feature-driven development. Upper Saddle River, NJ: Prentice Hall PTR, 2002. ISBN 01-306-7615-2.

PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programování. ISBN 80-722-6636-5.

PETERKA, Jiří. Sága rodů LAN a WAN: Architektura klient/server. In: EArchiv: Archiv článků a přednášek Jiřího Peterky [online]. Peterka, 1997 [cit. 2018-11-16]. Dostupné z: http://www.earchiv.cz/a708s600/a708s632.php3

PETERKA, Jiří. Báječný svět počítačových sítí. In: EArchiv: Archiv článků a přednášek Jiřího Peterky [online]. Peterka, 2006 [cit. 2018-11-16]. Dostupné z: http://www.earchiv.cz/b06/b0400010.php3

SHORE, James a Shane WARDEN. The art of agile development. Sebastopol, CA: O'Reilly Media, 2008. Theory in practice (Sebastopol, Calif.). ISBN 978-0-596-52767-9.

ZAPPONI, Carlo. GitHut - Programming Languages and GitHub [online]. 2014 [cit. 2018-09-24]. Dostupné z: https://githut.info/

Electron - Build cross platform desktop apps with JavaScript, HTML, and CSS. [online]. San Francisco [cit. 2018-11-24]. Dostupné z: https://electronjs.org/

GitHub [online]. San Francisco, 2008 [cit. 2018-11-25]. Dostupné z: https://github.com/

Seznam příloh